Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departmento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação
Doutorado em Ciência da Computação

Engineering Efficient Exception Handling for Android Applications

Juliana de Araújo Oliveira

Natal-RN
Maio 2021

Juliana de Araújo Oliveira

Engineering Efficient Exception Handling for Android Applications

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Linha de pesquisa:
Engenharia de Software

Orientador

Prof. Dr. Nélio Alessandro Azevedo Cacho

PPgSC – Programa de Pós-Graduação em Sistemas e Computação
DIMAp – Departamento de Informática e Matemática Aplicada
CCET – Centro de Ciências Exatas e da Terra
UFRN – Universidade Federal do Rio Grande do Norte

Natal-RN

Maio 2021

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Oliveira, Juliana de Araújo.
Engineering efficient exception handling for android applications / Juliana de Araújo Oliveira. - 2021.
152f.: il.

Tese (doutorado) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Departamento de Informática e Matemática Aplicada, Programa de Pós-Graduação em Sistemas e Computação. Natal, 2021.
Orientador: Prof. Dr. Nélio Alessandro Azevedo Cacho.

1. Computação - Tese. 2. Tratamento de exceção - Tese. 3. Android - Tese. 4. Robustez - Tese. 5. Visão holística - Tese. 6. Consumo de energia - Tese. I. Cacho, Nélio Alessandro Azevedo. II. Título.

RN/UF/CCET CDU 004

Elaborado por Joseneide Ferreira Dantas - CRB-15/324

Tese de Doutorado sob o título *Engineering Efficient Exception Handling for Android Applications* apresentada por Juliana de Araújo Oliveira e aceita pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Prof. Dr. Nélio Alessandro Azevedo Cacho
Presidente
DIMAp – Departamento de Informática e Matemática Aplicada
UFRN – Universidade Federal do Rio Grande do Norte

Prof. Dr. Windson Viana de Carvalho
Examinador
IUV – Instituto Universidade Virtual
UFC – Universidade Federal do Ceará

Prof. Dr. Fernando José Castor de Lima Filho
Examinador
CIn Centro de Informática
UFPE Universidade Federal de Pernambuco

Prof. Dr. Eiji Adachi Medeiros Barbosa
Examinador
IMD – Instituto Metrópole Digital
UFRN – Universidade Federal do Rio Grande do Norte

Prof$^{\underline{a}}$. Dr$^{\underline{a}}$. Roberta de Souza Coelho
Examinador
DIMAp – Departamento de Informática e Matemática Aplicada
UFRN – Universidade Federal do Rio Grande do Norte

Natal-RN, 31 de Maio de 2021.

Ao meu Deus, a minha família, principalmente minha mãe e a todos que trilharam esse caminho comigo.

Agradecimentos

O caminho para a finalização de um doutorado não é fácil, nem trivial. Passamos por muitos momentos difíceis, com pressão, estresse e cobranças, geralmente a pior cobrança vem de você mesmo. Também há momentos de felicidade e realização, frutos que vamos colhendo com o esforço e trabalho. E tem muito esforço, muita renúncia. Nesse caminho, encontramos o apoio de muitos e vamos superando cada obstáculo, as vezes pensamos que não vai dar, mas continuamos. No ano de 2020 aconteceu algo que ninguém esperava: uma pandemia que nos assola até agora. Não estávamos preparados para isso e finalizar este caminho se tornou muito mais pesado e difícil.

Nesse momento final olhamos para trás e vemos todo o caminho percorrido, todas as experiências adquiridas e apesar de ser um momento difícil, podemos agradecer. Eu tive excelentes experiências neste período de doutorado que vou levar para minha vida pessoal e profissional. Pude vivenciar um período de sanduíche na universidade de Newcastle Upon Tyne, UK, onde tive todo o apoio e colaboração do professor Alexander Romanovsky e do pesquisador Ashur Rafiev, bem no final deste caminho, que coincidiu com a pandemia, mas foi de grande valia para mim.

Feita esta breve reflexão, quero agradecer primeiramente a Deus por ter chegado aqui, a fé é algo que nos dá força para continuar. Quero agradecer a minha família que me deu todo o apoio como sempre, especialmente minha amada mãe que é meu suporte principal, com orações, carinho e palavras de ânimo. Agradeço a outra pessoa muito importante, meu orientador Nélio Cacho, que tem sido em todos esses anos um exemplo de pesquisador e pessoa humana, ele tem minha admiração. Guiar um aluno neste caminho não é tarefa fácil e ele sempre soube compreender todo o contexto e guiar na direção certa. Agradeço aos amigos que fiz e que ajudaram o dia-a-dia a ficar mais leve, não poderei citar todos aqui, mas tive muita sorte de tê-los. Romênia, Arthur e Larysse foram três desses amigos que sempre me fizeram muito bem.

Por fim, agora começa uma nova história e um novo caminho será trilhado. Vamos com fé, pois tudo passa.

Confia no Senhor de todo o teu coração e não te estribes no teu próprio entendimento. Reconhece-o em todos os teus caminhos, e ele endireitará as tuas veredas.

Provérbios 3:5-6

Engineering Efficient Exception Handling for Android Applications

Autor: Juliana de Araújo Oliveira
Orientador(a): Prof. Dr. Nélio Alessandro Azevedo Cacho

Resumo

A popularidade da plataforma Android pode ser atribuída à capacidade de executar aplicativos que potencializam os diversos recursos dos dispositivos móveis. Os aplicativos Android são em sua maioria escritos em Java, no entanto, eles são muito diferentes dos aplicativos Java padrão, com diferentes abstrações, vários pontos de entrada e também têm uma forma diferente de comunicação entre os componentes. Estas diferenças na estrutura de aplicações Android têm produzido efeitos negativos na experiência do usuário em termos de baixa robustez. Em termos de robustez, o mecanismo de tratamento de exceções para a plataforma Android tem dois problemas principais: (1) a abordagem *"Terminate ALL"* e (2) a falta de uma *visão holística* do comportamento excepcional. O tratamento de exceções está fortemente relacionado à robustez do programa. Além da robustez, o consumo de energia e o desempenho são outros requisitos não funcionais que precisam ser levados em consideração durante o desenvolvimento. Esses três requisitos podem afetar diretamente a qualidade da experiência do usuário e a qualidade do funcionamento dos aplicativos. Neste contexto este trabalho propõe uma metodologia geral para engenharia eficiente de aplicativos Android e um EHM denominado DroidEH para suportar a metodologia e melhorar a robustez de aplicativos Android. Estudos foram realizados para entender o impacto do tratamento de exceções na robustez e no consumo de energia dos aplicativos Android. A avaliação da metodologia mostrou que ela é satisfatória para atingir o objetivo de permitir ao desenvolvedor tomar decisões levando em consideração esses requisitos não funcionais e determinar através do trade-off entre esses requisitos, diferentes modos de operação que podem ser implementados no aplicativo usando o DroidEH. Além disso, foi observado que uso do DroidEH em aplicativos pode aumentar sua robustez.

Palavras-chave: Tratamento de Exceção, Android, Robustez, Visão Holística, Consumo de Energia.

Engineering Efficient Exception Handling for Android Applications

Author: Juliana de Araújo Oliveira
Supervisor: Prof. Dr. Nélio Alessandro Azevedo Cacho

Abstract

The popularity of the Android platform can be attributed to their ability to run apps that leverage the many capabilities of mobile devices. Android applications are mostly written in Java, however they are very different from standard Java applications, with different abstractions, multiple entry points, and also have a different form of communication between components. These differences in the structure of Android applications have had negative effects on the user experience in terms of low robustness. In terms of robustness, the exception handling mechanism for the Android platform has two main problems: (1) the *"Terminate ALL"* approach and (2) a lack of a *holistic view* on exceptional behavior. Exception handling is strongly related to program robustness. In addition to robustness, energy consumption and performance are other non-functional requirements that need to be taken into account during development. These three requirements can directly affect the quality of the user experience and the quality of the functioning of the applications. In this context this work proposes a general methodology to efficient engineering of Android applications and an EHM called DroidEH to support the methodology and to improve the robustness of Android applications. Studies have been carried out to understand the impact of exception handling on the robustness and energy consumption of Android applications. The evaluation of the methodology showed that it is satisfactory in achieving the objective of allowing the developer to make decisions taking into account these non-functional requirements and to determine through the trade-off between these requirements, different operation modes that can be implemented in the application using the DroidEH. Furthermore, it was observed that the use of DroidEH in applications can enhance their robustness.

Keywords: Exception Handling, Android, Robustness, Holistic View, Energy Consumption.

List of Figures

List of Tables

List of Acronyms

ART – Android Runtime

GUI – Graphical User Interface

UI – User Interface

EHM – Exception Handling Mechanisms

EHC – Exception Handling Context

API – Application Programming Interface

HAL – Hardware Abstraction Layer

JVM – Java Virtual Machine

HFT – Holistic Fault Tolerance

FT – Fault Tolerance

OM – Operation Mode

LOC – Lines Of Code

GPS – Global Positioning System

EH – Exception Handling

APK – Android Package

SIM – Simulation File

NFC – Near-field Communication

QR – Quick Response

IDE – Integrated Development Environment

List of Symbols

Δ (Delta)

$\sum$ (Sum)

Contents

1 Introduction

According to a recent report by Global Stats (StatCounter Global Stats, 2017), the Android platform has overtaken Microsoft Windows for the first time as the world's most popular operating system in terms of total internet usage across desktop, laptop, tablet and mobile combined. The popularity of the Android platform can be attributed to their ability to run apps, end-user programs that leverage the many capabilities of these devices, potentially in unforeseen ways. Apps are in widespread use. According to (STATISTA, 2021), the Google Play store reached 3.04 million available apps until last year and 80.6 billion app downloads since its creation.

Robustness, the ability of a program to properly cope with errors during its execution (LEE; ANDERSON, 1990), is an important attribute to users of mobile apps. Apps that frequently exhibit functional errors or crash tend to have bad ratings in app stores (KHALID et al., 2015). In most of the languages employed in app development, exception handling is the primary approach to signal and handle the occurrence of errors at runtime (ALBAHARI, 2012; CABRAL; MARQUES, 2007; PHILLIPS; HARDY, 2013). Thus, exception handling is strongly related to program robustness, and the way developers use exception handling has a potentially strong impact on the robustness of any application. In Java, the language in which most Android applications are written, a program can indicate that an error occurred by throwing an exception and attempt to recover by handling ("catching") that exception and implementing the code with the proper approach to handle the exception. If an unchecked exception is thrown but not handled, the program may crash. Android applications use this same mechanism to report and handle errors.

In addition to robustness, energy consumption and performance are other non-functional requirements that need to be taken into account. These two requirements may directly affect the quality of the user experience and the quality of the functioning of the applications. Studies have shown that battery consumption is among the main complaints of users (KHALID et al., 2015; MAN et al., 2016), related to applications draining battery. Regarding performance, many studies try to point out better ways for developers to improve the performance of their applications (LINARES-VASQUEZ et al., 2015; LEE; LEE, 2011; HECHT; MOHA; ROUVOY, 2016; LIU; XU; CHEUNG, 2014). Like exception handling issues, problems related to energy consumption and performance can lead to the malfunction of applications and generate disinterest on the part of users.

Although Android apps are written in Java, they present differences from standard Java applications (or simply "Java applications", i.e., those not targeting Android). Android apps are organized in terms of abstractions that are not available for Java applications, also Android applications have multiple entry points related to lifecycle methods and event-handlers methods. Moreover, they typically make use of constructs for asynchronous programming (INC., 2015; CHIN et al., 2011). These constructs aim to improve app responsiveness and are distinct from Java's constructs for concurrent programming.

All the Android-specific abstractions can throw exceptions, with different effects. For example, an `ActivityNotFoundException` thrown in a call to the `startActivity` method can be caught similarly to any Java exception. However, if an exception is thrown during the execution of a started activity, the only way of handling it is by creating an instance of `UncaughtExceptionHandler` and associating it with the GUI thread. Otherwise, these exceptions will always cause the application to crash ("Force Close" in Android terminology). Both asynctasks and intent-related calls can end their execution raising an exception.

Previous studies (COELHO et al., 2015, 2017; WU et al., 2017; OLIVEIRA et al., 2018) have pointed out problems related to the robustness of Android apps and lack of robustness from users' perspective (KHALID et al., 2015; MAN et al., 2016; LIM et al., 2015). These studies serve as a motivation for this work since their results show that even using the Java exception handling mechanism, Android apps have robustness problems related to the platform or the misuse of the mechanism by developers. In this sense, we argue that the exception handling mechanism provide by the Android platform has introduced two main problems for application developers: (1) the *"Terminate ALL"* approach and (2) the lack of a *holistic view* on exceptional behavior.

In the context of energy consumption, some studies have investigated the energy consumption of specific device components like Screen, GPS, Network, and others investigated that at the code level. To the best of our knowledge, no study has set out to investigate whether the use of exception handling strategies has an impact on energy consumption of Android applications. To better understand this, a study was carried out by performing experiments using ten Android applications. The results showed that it is important to evaluate the impact on energy consumption at the exception handling level during the development of Android applications.

Hence, the main objective of this research is to provide an engineering efficient exception handling methodology and tools for Android applications through an interactive process. Those proposed solutions aim to help developers in making informed decisions about when and how exception handling code should be used. This decision should take into account non-functional requirements, such as robustness, energy efficiency and performance which are relevant concerns for both users (KHALID et al., 2015; MAN et al., 2016; LIM et al., 2015; Wilke et al., 2013) and developers(MANOTAS et al., 2016) of mobile applications.

To support the proposed methodology, a new exception handling mechanism, DroidEH, has been proposed to improve the exception handling and robustness of Android applications. DroidEH mechanism is grounded on two main approaches: the *EFlow* model (CACHO et al., 2008; CACHO; COTTENIER; GARCIA, 2008) and the concept of *Holistic Fault Tolerance*(GENSH et al., 2017a). The proposed methodology and DroidEH mechanism were evaluated to assess its feasibility to improve robustness of Android applications.

1.1 Problem Statement

Most of the Android apps are written in Java. Each app runs within a separate process in the Android Runtime (ART)[1], which means that memory spaces are not shared between apps. Since it is common for apps to require services provided by other apps, for example, to share a photo taken by the Camera app using Whatsapp, Android provides standard facilities for apps to communicate without having to share memory.

The Android application framework defines a number of basic components that are not available in the general Java development (INC., 2015; CHIN et al., 2011). *Activities* interact with users by means of a GUI. *Services* execute long-running operations in the background and without user interaction. *Broadcast receivers*, on the other hand, work as listeners, registering their interest in events that will be sent by the platform or other components as background broadcast messages. One important concept of the platform is the `Intent` object. It represents a message to be sent to the platform in order to request the execution of other components. It is used for communication between com-

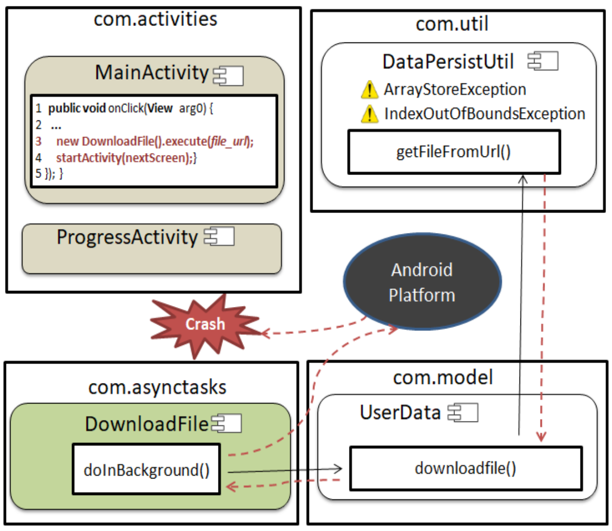
[1]ART was introduced in Android 4.4, released in Sept 2013, and completely replaced the Dalvik runtime in Android 5.0, released in Jun 2014.

ponents, to start an activity or service asynchronously, or to deliver broadcast messages. Finally, *AsyncTasks* are used to execute short background operations asynchronously, in order to avoid blocking the application UI thread. Method calls to Android abstractions may throw exceptions. Exception handling mechanisms (AL., 2001; GOODENOUGH, 1975; PARNAS; WURGES, 1976) (EHM) are the most common approach to coping with errors in the development of software systems. Android applications written in Java inherit the EHM of Java and this mechanism has introduced two main problems for the developers. This section presents these two problems that are related to the robustness of Android applications and problems related to other non-functional requirements.

Terminate ALL approach: According to Cui and colleagues (CUI et al., 2015), one of the main differences between traditional Java programs and Android applications is that traditional Java programs use a single method (i.e, the `main` method) as the entry point whereas Android components can have many entry points. These entry points include Android lifecycle methods (*onCreate*, *onStart*, *onReceive* etc.) and user-defined event handlers. According to the Android platform documentation[2], there are 18 entry point methods in `Activity`, 6 in `Service`, 5 in `AsyncTask` and 1 in `BroadCast`. All those methods are invoked by the Android framework at runtime and their order of execution cannot be determined in advance (CUI et al., 2015). As a consequence, when a thrown exception reaches one of those methods and cannot be caught by any prepared catcher, it would finally trap into the *uncaughtException* function of the Android exception handling mechanism (WU et al., 2017). Wu and colleagues(WU et al., 2017) perceived that the *uncaughtException* function kills the exceptional process straightforwardly regardless of the process attribute (WU et al., 2017). This *"Terminate ALL"* approach kills even critical system services. When that happens, the Android system is crashed and rebooted (WU et al., 2017).

To illustrate this problem, Figure 1 depicts a hypothetical Android application that downloads free music from free download sites. This application comprises five classes: two activities `MainActivity` and `ProgressActivity` extending `android.app.Activity`, one asynctask `DownloadFile` extending `android.os.AsyncTask`, and two classes implementing the business rules. Initially, the Android platform creates an instance of `MainActivity`, invokes the `onCreate` method to show the available mp3 songs, and register a button on the main application screen. When the user clicks the button, the `onClick` method downloads the selected song by making two asynchronous calls to: create the `DownloadFile` AsyncTask to download a file and to start the `ProgressActivi-`

[2]https://developer.android.com/guide/topics/ui/ui-events.html



```
public void onClick(View  arg0) {
 ...
   new DownloadFile().execute(file_url);
   startActivity(nextScreen);}
}); }
```

Figure 1: Example of Android application and exception propagation. (OLIVEIRA et al., 2018a)

`ty` to show a progress bar whilst the song is downloaded.

Android accepts all these requests and changes the current UI screen by stacking `Progress-`
`Activity` on `MainActivity`, calling `ProgressActivity's onCreate` method. In another thread, the `doInBackground` method invokes the `downloadfile` method. This method invokes `getFileFromUrl` that throws an exception. The EHM performs at runtime the search for a proper handler. However, this search is limited to the memory space of the current *AsyncTask*. For instance, in the hypothetical scenario, the unchecked exception `IndexOutOfBoundsException` propagates up the call stack(to method `downloadfile`) until a matching handler is found. If no handler is found in the `doInBack-groud` method, exception `IndexOutOfBoundsException` is propagated to the Android platform. In this scenario, the whole application will be terminated.

To complicate the situation, the Android platform intensively employs unchecked exceptions (COELHO et al., 2015; OLIVEIRA et al., 2016). For instance, Coelho and colleagues (COELHO et al., 2015) analyzed 6.005 exception stack traces reported on 482 Android projects hosted in Github and 157 projects hosted in Google Code. They observed that 64.85% of the reported exceptions are unchecked and 50% of the catch blocks that convert any checked exception into an instance of `RuntimeException` are performed by methods defined on the Android platform. For example, according to the documentation of class Activity[3], 12 public methods may throw `SecurityException`, 10 may throw `ActivityNotFoundException`, and 6 may throw `SendIntentException`.

[3]http://developer.android.com/reference/android/app/Activity.html.

Among these exceptions, only the latter is checked. Any other exception thrown by these methods is necessarily also unchecked.

In languages such as C#, where the compiler does not perform checks related to exceptions, developers are used to being extra careful in order to avoid uncaught exceptions. However, in Java, developers rely on the compiler pointing out when they forget to catch an exception or explicitly include it in the signature of a method. The problem is that, in Android app development, since the Android platform throws unchecked exceptions most of the time, the compiler does not perform these checks, which leads to uncaught exceptions becoming a common cause of bugs in Android applications (COELHO et al., 2015; OLIVEIRA et al., 2016, 2018).

To alleviate these problems, the Android runtime attaches a process global `Uncaught-ExceptionHandler` when an application is started (GÖRANSSON, 2014). The `Uncaught-ExceptionHandler` interface is used by implementing method `uncaughtException` and attaching it to an *Activity* or the whole application. If the activity/application is terminated due to an exception, the handler is invoked on the terminating activity before it terminates. The main problem with this approach is that it is not possible to avoid terminating the application abnormally [4]. Hence, this handler offers a chance for the application to complete its execution gracefully, or at least to make a note of the error to a network or file resource. After analyzing 112 versions of 16 Android projects, Oliveira and colleagues (OLIVEIRA et al., 2018) could not find any class implementing the `UncaughtExceptionHandler` interface. This result may indicate that this interface is not really helping with handling exceptions properly.

Lack of Holistic View of Exceptional Behavior: Holistic fault tolerance (GENSH et al., 2017a) is a general approach to architecting system fault tolerance in a way that is more suitable to the efficient and effective implementation of fault tolerance than the traditional approach when fault tolerance is associated with system structuring (e.g. out of layers, classes, components, etc.). It simplifies and modularises a cross-cutting development of fault tolerance in a resource efficient way. The idea is to support a flexible choice of which part of the system and how to involve in recovery after an error is detected, and to directly connect the source of an error with the corresponding recovery measures.

As shown in Figure 1, the Android platform provides an exception handling approach that is inherently local to a problem that is intrinsically global. It means that there is *no holistic view on the exceptional behaviour* and the design and implementation of ex-

[4] http://bit.ly/1iR1tU3

ception handling code focus on the method level, rather than being application-wide. It is not possible in Android applications to monitor all started activities, services, broadcast receivers and asynctasks to handle exception occurrences and orchestrate the exception handling actions. For instance, it would be useful to detect the occurrence of exceptions `IndexOutOfBoundsException` and `ArrayStoreException` in the aforementioned example (see Figure 1) and retry only the execution of `DownloadFile` and `ProgressActivity`, rather than crashing the whole application. In another scenario where the `DownloadFile` asynctask is running normally and it is not possible to retry the execution of `ProgressActivity`), the application should be able to run normally since its main purpose is to download songs.

In order to implement an application with that, exception handling constructs (`try-catch` blocks) should be used to handle exception occurrences at all entry points, event listeners, and event handler methods. In addition, the error handling code has to be defined by means of a separate component that the exception handlers invoke. This solution is not ideal, as it imposes an implementation overhead (due to the implementation of try-catch blocks) and requires the actual error handling code to be implemented outside of the exception handlers. Moreover, scattering and tangling the exceptional code cause well-known maintainability side effects (ROBILLARD; MURPHY, 2007; CACHO et al., 2014, 2014) since they makes it difficult to achieve explicit separation of exceptional behaviour (LIPPERT; LOPES, 2000).

This lack of holistic view of the exceptional behaviour provided by the Android platform stems from the fact that traditional exception handling mechanisms only provide constructs for (re-)raising and handling exceptions (ROBILLARD; MURPHY, 2000, 2003; **??**). However, not much support is provided to the task of understanding and maintaining the exception paths. To understand the possible paths of each single exception, a programmer usually needs to both: (i) memorise all its exception supertypes, and (ii) exhaustively examine all the interfaces of modules traversed by that exception - from components which raised it to all the potential points where they are handled. Analyzing such exception paths becomes even more time-consuming in the Android platform where components are loosely coupled, and exceptions not handled locally can crash the whole application. This is why programmers usually introduce faults in the implementation and evolution of exceptional behaviour even in the presence of Java's reliability checks (COELHO et al., 2017; OLIVEIRA et al., 2016, 2018). A classical example occurs when a component may be changed to raise additional exceptions whereas other unchanged modules have to handle them by using existing handlers (OLIVEIRA et al., 2016).

Other Non-functional Requirements: In addition to robustness, energy consumption and performance are other non-functional requirements that need to be taken into account during development and that directly affect the quality of the user experience and the quality of the functioning of the applications. Some studies show battery consumption is among the main complaints of users (KHALID et al., 2015; MAN et al., 2016) related to applications draining battery. Many studies are dedicated to investigating energy consumption in mobile applications for specific device components like GPS, Network, and at the code level (LI et al., 2014, 2013; OLIVEIRA; OLIVEIRA; CASTOR, 2017; LINARES-VÁSQUEZ et al., 2014; CHOWDHURY et al., 2018; ZHANG; MUSA; LE, 2013). Regarding performance, many studies try to point out better ways for developers to improve the performance of their applications. The study by Linares-Vasquez, Mario, et al.(LINARES-VASQUEZ et al., 2015) investigates how developers detect and solve performance problems. The study of Lee, Jae Kyu, and Jong Yeol Lee (LEE; LEE, 2011) presents techniques that can help improve performance. Other studies try to investigate problems related to performance, like the study by Hecht, Geoffrey, Naouel Moha, and Romain Rouvoy (HECHT; MOHA; ROUVOY, 2016) that analyzes the impact of Android code smells on performance and the study by Liu, Yepang, Chang Xu, and Shing-Chi Cheung (LIU; XU; CHEUNG, 2014) that presents a tool that detects and characterizes performance bugs, this study considers UI lagging, high energy consumption and Memory bloat as bugs related to performance.

Problems related to energy consumption and performance can lead to the malfunction of applications and generate disinterest on part of users. The work of (LIM et al., 2015) points out that users from Brazil and Spain are two times more likely than other countries to stop using an app because it crashes and users from Brazil are also two times more likely than other countries to stop using an app because it is slow, for example. The focus of this work is to present a methodology that allows the developer to carry out a trade-off between these requirements during the design of their application, in order to build a more efficient and reliable application. Studies and evaluations presented in the scope of this work emphasize the impact of the exception handling on the robustness and energy consumption of Android applications, which are the most common problems pointed out by users.

1.2 Goals

This thesis aims to provide a methodology and tools to engineering efficient exception handling for Android applications. To achieve this main objective it is necessary to reach the following secondary objectives:

1. Perform studies to better understand the impact of exception handling on the robustness in Android applications and the impact of exception handling on energy consumption of Android applications;

2. Propose a methodology to engineering efficient exception handling for Android applications;

3. Propose an exception handling mechanism for Android applications to improve the exception handling and robustness of the applications that afford the methodology proposed;

4. Evaluate the proposed methodology and exception handling mechanism;

1.3 Research Questions

The goals presented in the previous section aim to answer some research questions of this work. The main research question that guides this work is presented below:

Research Question: *How to efficiently implement exception handling code for Android applications taking into account the trade-off between robustness and other non-functional requirements?*

This question can be splitted into three specific questions:

- **Question 1 (RQ1)** *What is the impact of using Android exception handling on application robustness during evolution?*

  The purpose of this question is to evaluate the exception handling during the evolution of Android applications, analysing changes of normal and exception handling code and executing exception flow analysis, to assess whether there is an impact of exception handling, in relation to Android abstractions, on the robustness of applications.

- **Question 2 (RQ2)** *The use of exception handling strategies in Android applications can impact on their energy consumption?*

  This question investigate the impact of using some exception handling strategies in Android applications on the energy consumption. The strategies were inserted on lifecycle methods and event-handlers methods of the study applications and experiments were performed using the Monkey tool, that generates random events in the user interface. The energy data were collected using `batterystats` tool from Android and visualized Battery Historian tool to further analysis.

- **Question 3 (RQ3)** *Is it possible to implement exception handling code and make decisions that take into account non-functional requirements in order to obtain a more robust and energy-efficient Android Application?*

  The purpose of this question is to evaluate if the proposed methodology can be applied in the development of Android applications, using the tools that support it. Using the methodology, the developer must be able to implement exception handling in the application to make it more robust while also taking into account its energy consumption. Based on the methodology, the developer can make decisions that will be reflected in the implementation, generating different modes of operation for the application, which seek to achieve a balance between these non-functional requirements.

1.4 Thesis Contributions

After reaching the objectives specified in this research, answering the research questions described in the previous section, this work presents five main contributions:

1. **An empirical study to analyze the relationship between the usage of Android abstractions and uncaught exceptions (OLIVEIRA et al., 2018)**, involving:

   - Analysis of changes to both normal and exception handling code (or "exceptional code", the `catch` blocks) in 112 versions extracted from 16 software projects covering a number of domains, amounting to more than 3 million LOC;
   - Change impact analysis and exception flow analysis of the projects during evolution.

2. **An empirical study to investigate whether the use of exception handling strategies impact on the energy consumption of Android applications**, involving:

- Selection of ten Android application projects;
- Definition of three exception handling strategies to be used in applications;
- Insertion of the strategies in the code of the applications for lifecycle methods like `onCreate` and event-handlers methods like `onClick`;
- Definition of experiments execution using Monkey tool;
- Execution of 10 experiments without exception handling strategies and 30 experiments with the exception handling strategies.

3. **A comprehensive methodology that supports trading off power consumption, reliability and resource usage during exception handling design**, involving:

   - Definition of the proposed workflow in terms of: manual activities and automatic activities, the sequence of these activities and the tools to support the methodology;
   - Implementation of the automatic model generation from an Android application in order to perform the simulations;
   - Implementation of the static analysis to calculate exception probabilities at three levels: application level, classes or components level, and method level.

4. **A new exception handling mechanism for Android applications (OLIVEIRA et al., 2018a; OLIVEIRA, 2018)**, involving:

   - Defining the conceptual model;
   - Defining the mechanism abstractions;
   - Implementation of the new mechanism;

5. **Evaluation of the proposed methodology and exception handling mechanism, DroidEH**, involving:

   - Definition of the target applications;
   - Analysis of the applications to identify the main functionalities, components, and interactions;
   - Preparation of the applications for the experiments, adding code to allow the continuous execution of the components;
   - Execution of the characterization experiments. A total of 36 experiments were carried out;

- Applying the characterization experiments results on the simulation files;
- Application of the methodology on the target applications;
- Definition of study hypothesis for assessment of DroidEH;
- Selection of the subjects and target applications;
- Definition and execution of study phases;
- Analysis of the study results and questionnaire Data.

1.5 Thesis Structure

The remainder of this text is organized as follows. Chapter 2 presents the basic terminology related to exception handling mechanisms, a overview about Android platform, static analysis, EFlow model and Holistic Fault Tolerance concept. Chapter 3 presents two studies carried out to understand the impact of exception handling on the robustness and energy consumption in Android applications. The first is a maintenance study performed in order to identify problems related to exception handling in Android applications throughout evolution of applications, the chapter shows de experimental procedures, results and analysis of study. The second study was performed to evaluate the impact of exception handling strategies on energy consumption of Android applications, the chapter shows the study setting, details about the experiment execution and the results. Chapter 4 shows in detail the proposed methodology to engineering efficient exception handling for Android applications, showing the modelling and characterization experiments and a novel exception handling mechanism for Android applications, detailing the conceptual model, proposed abstractions and implementation. Chapter 5 presents the evaluation of the methodology proposed and of the exception handling mechanism. Finally, chapter 6 presents the related works and Chapter 7 shows the final remarks of the thesis.

2 Background

This chapter presents the concepts and definitions discussed in this dissertation. Section 2.1 deals with the concepts of fault, error and failures. Section 2.2 shows the concepts of exception and exception handling models. Next, an overview of exception handling mechanisms is presented in section 2.3. In section 2.4 the platform *Android* is presented, detailing its architecture and components and in section 2.5 static analysis concepts are presented. Section 2.6 presents the Eflow model and its implementations. Finally, section 2.7 shows the concept of Holistic Fault Tolerance.

2.1 Faults, Errors and Failures

A software system can be seen as a set of components that cooperate to meet the demands of the system environment. These components are under the control of a design, that can be considered as a special component, which is responsible for defining the interactions between the components and establishing the connections between them and the environment (LEE; ANDERSON, 1990).

In the work of Avizienis, Algirdas, et al. (AVIZIENIS et al., 2004), an environment is defined as other systems, which include, hardware, software, humans, and the physical world with its natural phenomena. A system provides a service to the environment, this service is the behavior of the system in the view of its users, i.e., what the system does to implement its function and is described by a sequence of states. When the system performs its function correctly, it performs valid transitions between states, going from one valid internal state to another. However, erroneous situations may occur during the execution of a system. They may be caused by one or more components or by the system design itself.

In this way, the following concepts can be defined: A failure occurs when the service offered by the system does not correspond to a correct service. It can be seen as an event that occurs when the service does not match what is defined in the system specification. Failure is perceived externally by system users; As a service is a sequence of states, in erroneous situations an error can be defined as a part of the internal state of the system that is susceptible to lead to a defect.

And in turn, fault is the hypothetical cause of an error. A fault may originate internally by the system or some of its components, or externally to the system, through interactions that lead to error. For example, the user provides an invalid entry for the system. Figure 2 shows the sequence of occurrence of a fault until the manifestation of failure in the system.

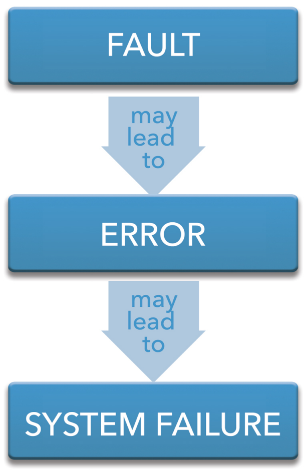


Figure 2: Sequence for fault, error and failure.

2.2 Exceptions and Exception Handling Models

As was explained in section 2.1, a defect can lead to an error, which in turn can lead to a failure. These concepts can be exemplified in the context of Android applications as follows: a user accesses an application that functions as a calculator and provides an invalid entry, which is not handled internally. One of the internal components of the application detects the problem and immediately the application ends the execution with a force close screen, visually informing the user that a problem has occurred.

In any computer system, erroneous situations can occur, exception handling (GOODENOUGH, 1975) is a technique commonly used by leading programming languages, which provides means of detection, signaling and error handling. The purpose of using this technique is that applications can be more robust and reliable. Exception handling should allow a system to be able to react appropriately to exceptional conditions, ensuring the integrity of the current state of the system. In this way, it can either terminate execution or allow it to continue from a valid state after handling the error (LEE; ANDERSON, 1990).

Situations such as the example cited in this section are defined in Goodenough, John B. (GOODENOUGH, 1975) as exceptional conditions that must be communicated to an invoker. That is, in the presence of exceptional conditions, an exception must be thrown to the invoker and it must respond to that event by handling the exception. According to Miller and Tripathi (MILLER; TRIPATHI, 1997), an exception is an abnormal computation state, and an exception instance is an instance of an exception. In general, an exception is seen by developers as an error.

As previously mentioned, a system can be viewed as a set of components, each component can receive service requests and must return responses. In some situations a component may not be able to perform a service correctly, in which case it should return an exception. In this way, a component can have two types of response: normal and exceptional.

Regarding the exceptional response of a component, exceptions can be classified into three categories: (i) interface exceptions, are flagged in response to a request that does not conform to the component interface; (ii) failure exceptions, which are flagged when, for some reason, the component can not provide the requested service; (iii) internal exceptions, which are exceptions thrown by a component that wants to invoke its own internal fault tolerance mechanism (LEE; ANDERSON, 1990).

In this context, an ideal fault-tolerant component, Figure 3, is a component that implements internal mechanisms for handling exceptions. The activity of this component can be divided into two: normal and abnormal (or exceptional). Normal activity is the part of the component responsible for implementing your normal services, all part of business logic. The exceptional part implements the necessary measures to handle exceptions. At each system level, the components treat the exceptions that were thrown during normal activity or exceptions that were thrown by lower-level components of the system. If an exception can not be handled in the component that launched it, it is flagged for the component of an upper layer.

Exceptions may occur during the execution of a system, and each component of that system must function as an ideal fault-tolerant component capable of correctly handling erroneous situations. This ensures that the system always returns to a valid state.

In the scenario of an exception occurring, the component should generally be aware

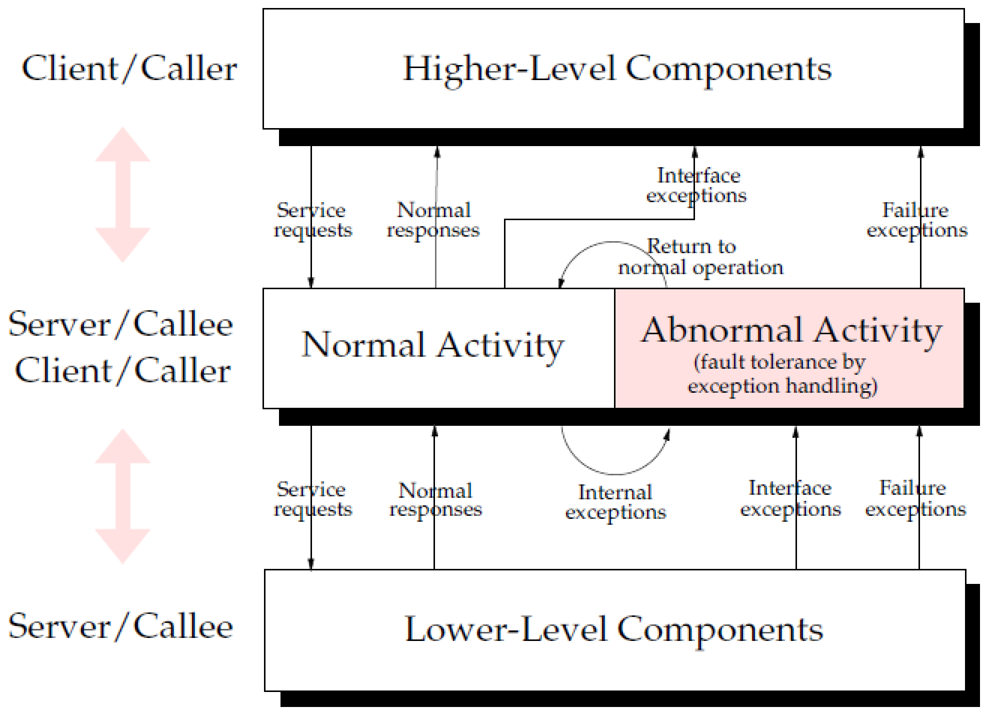


Figure 3: Ideal Fault Tolerant Component. (LEE; ANDERSON, 1990)

of where this exception originated, who will be the recipient of the exception, and how the treatment will be done. At Miller, Robert, and Anand Tripathi (MILLER; TRIPATHI, 1997), the exception target is defined as the receiver of the notification and responsible for setting the handling, the signaler is the location in the source code or the component that generated the exception.

Once the target is notified of an exception occurring, it must search for appropriate handlers to handle it. An exception handler is the code invoked in response to an exception occurrence. The search starts with the handlers associated with the target and an exception is considered handled when the execution of the handler is completed and the system control flow returns to the normal state. An exception context is the available information that the handler knows about the occurrence of the exception.

In many cases, an exception thrown by the signaler is not handled at its exception target, in this case, the exception must be flagged for the target invoker, we call this exception propagation. There are two ways to do this: implicit, the exception is automatically flagged and is of the same type as the original exception; explicit, in this case the exception must be explicitly thrown by the target handler and may be of a different type than the original.

An exception handling model defines the interactions between the signaler, exception target, and the handler. Some types of models known (MILLER; TRIPATHI, 1997) in the literature are: termination model, this model automatically finalizes the execution of the signaler, destroying any object in its scope and considers the target as a new signaler of the invoker of the operation; resumption model, computation continues from the point where the exception was originally thrown; retry model, in this model, when the exception is handled, the code block of the signaler is finalized and invoked; replacement model, is a variant of the termination model, the block of code that originated the exception is terminated and the result of the handler is returned to the caller's signaler.

2.3 Overview of Exception Handling Mechanisms

Many programming languages support exception handling mechanisms (C++, Java, C#, F#, VB.NET). These mechanisms allow developers to define exceptions, explicitly throw exceptions, and structure exceptional component activity through the use of handlers. Each language defines a specific set of abstractions to represent exceptions and handlers.

The exception handling mechanism is responsible for changing the normal control flow of a program to the exceptional flow of control when an exception throw occurs (GARCIA et al., 2001). It must conform to the characteristics of the language, have a simplified design, provide means that abstract the complexity of handling exceptions, and provide security (ROMANOVSKY; SANDÉN, 2001).

The Java programming language supports exception handling, for example, by providing abstractions `try-catch-finally`. With these abstractions the developer can delimit the region of the code being protected by a handler within a `try` block, identify the exception type of the handler, and execute the code responsible for handling the exception. If the exception is caught, this part is defined within the `catch` block. The code inside the `finally` block always runs. The language also provides a means for the developer to explicitly throw an exception by using `throw` and means so that it can define exceptional interfaces.

The following will describe some concepts that are tied to the concept of exception handling mechanism in the context of the Java programming language and therefore in the context of Android application development.

2.3.1 Class Hierarchy

In object-oriented programming languages, exceptions are represented as classes. They provide a means for the developer to identify the type of problem that occurred during execution. Figure 4 shows the Java language exception hierarchy, where all exception classes inherit from class `java.lang.Throwable`.

Exceptions in Java can be classified as checked or unchecked. Checked exceptions inherit from class `java.lang.Exception`. These exceptions must always be declared in the exception interface of the method, using `throws`, or they must be handled by the target method. Unchecked exceptions inherit from class `java.lang.RuntimeException`. The existence of this type of exception occurs because of exceptional conditions that occur as a result of running an application in Virtual Machine. The `Error` class represents exceptions that occur due to internal errors in Java's runtime. Developers can create their own exception classes that inherit from any of these supertypes.

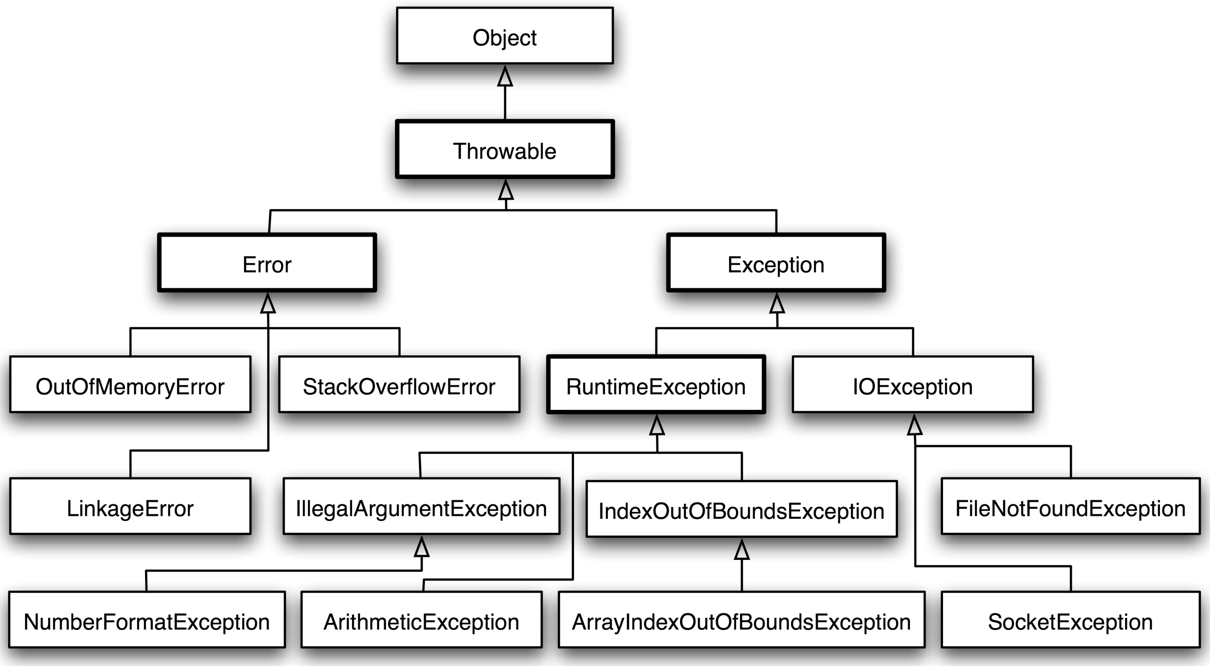


Figure 4: Class Hierarchy in Java. (GARCIA et al., 2001)

The type subsumption occurs when a value of a type A can be seen as a value of a supertype B (ROBILLARD; MURPHY, 2003). In the context of exceptions and their representation through classes, we can say that a handler can handle an exception by subsumption. For example, looking at Figure 4, if a method throws the `IllegalArgumentException` exception and the handler is of type `RuntimeException`, that handler can handle this exception, since the first exception inherits from the more generic exception .

2.3.2 Exception Interface

The caller of a method needs to know which exceptions may cross the boundary of the called one. In this way, the caller will be able to prepare the code beforehand for the exceptional conditions that may happen during system execution. For this reason, some languages provide constructs to associate to a method's signature a list of exceptions that this method may throw. Besides providing information for the callers of such method, this information can be checked at compile time to verify whether handlers were defined for each specified exception. This list of exceptions is defined by Miller and Tripathi (MILLER; TRIPATHI, 1997) as a method's *exception interface*. Ideally, the exception interface should provide complete and precise information for the method user.

However, they are most often neither complete nor precise (CABRAL; MARQUES, 2007), because languages such as Java provide means to bypass this mechanism. This is achieved by throwing a specific kind of exception, called *unchecked exception*, which does not require any declaration on the method signature. In fact, Java supports two type of exception: *checked* or *unchecked*. The compiler forces checked exceptions thrown within a method to be either associated with a handler or explicitly defined in the exception interface of that method. If a consumer method does not handle the exceptions specified in the exception interface of a provider method, nor specifies them in its own exceptional interface, compilation errors occur. On the other hand, the Java compiler does not require unchecked exceptions to be handled or to appear in exception interfaces.

2.3.3 Attachment of Handlers

A protected region is a domain that specifies the region of the computation where, if an exception is detected, a bound handler in that region will be activated (GARCIA et al., 2001). Regions can also be called Exception Handling Contexts (EHC). In a practical context, for example, a region can be a code block of a method or a single declaration within the method, where an exception can be thrown and protected by `try ... catch` in Java.

Handlers can be linked to different types of regions, such as: a statement, a block of code, a method, an object, a class, or an exception (GARCIA et al., 2001). Binding of handlers to declarations occurs when the handler is bound to a particular statement in the method body. Bindings to a block of code occur through language constructs that delimit the set of statements that will be protected by the handler. Languages like Java, C++, and C# use construct `try` to delimit the region.

A handler may be bound to a method and when an exception is thrown in one of the declarations of this method, the handler bound to it is activated. Binding a handler to an object or a class can occur by binding the handler to all declarations of that object or to all instances of a class. Handlers bound to an exception are bound by a method in the exception class and are activated when no more specific handler is encountered.

2.3.4 Handler Binding

In many languages, the search for the handler to deal with a raised exception occurs along the dynamic invocation chain. This is claimed to increase software reusability, since the invoker of an operation can handle it in a wider context (MILLER; TRIPATHI, 1997). The Android platform leverages Java's EHM and uses the `try-catch` constructs to handle and bind exceptions, as depicted by the following structure:

$$\texttt{try S}$$

$$\texttt{catch (}E_1\texttt{ x) }H_1$$

$$\texttt{catch (}E_2\texttt{ x) }H_2$$

The `try` block delimits a sequence S of statements that are protected from the occurrence of exceptions. The `try` block defines the normal code, and it is associated with a list of `catch` blocks. A `catch` block defines a sequence H_n of statements that implement the handling actions responsible for coping with an exception. `Catch` blocks are also called exception handlers. Each `catch` block has an argument E_n, which is an exception type. These arguments are filters that define what types of exceptions that `catch` block can handle. When a `catch` block defines as argument an exception type E_1, it can handle exceptions of type E_1 and exceptions that are subtypes of E_1.

There are three different types of workarounds that can bind handlers to occurrences of exceptions (GARCIA et al., 2001): (i) static approach, a handler is statically linked to a protected zone and is used for all occurrences of the exception while executing that region; (ii) dynamic approach, in this type of approach the connection will depend on the control flow of the program, ie, the handler that must be used is determined at run time, in the static approach, there is no such type of time search execution; (iii) semi-dynamic approach, is a combination of the two previous, local handlers can be statically linked to the regions. If a handler is not bound for some exception thrown in that region, a dynamic fetch is made to find an appropriate handler for the exception.

2.3.5 Propagation of Exceptions

Using exception handling mechanisms, a developer can define exceptions, throw exceptions and handle exceptions inside the application code. When an exception is not handled inside a method, this exception needs to be propagated until find a proper handling. According (AL., 2001) there two ways to do this propagation: explicit or automatic propagation. In the explicit propagation the developer needs to catch the exception and explicitly signal this exception along the call chain. The automatic propagation is usually adopted as the default behaviour and the developer don't need to signal explicit the exception to ensure that it will be propagated. Java language supports the two approaches. The explicit propagation can be used through the `throw` statement and the automatic propagation is supported by means of the use of exceptional interfaces in the methods signature.

2.4 Android Platform

Android is an open source platform that includes: a Linux-based operating system, the Hardware Abstraction Layer (HAL) that provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework, a set of C/C++ libraries used by various components, a Java API framework with platform-specific components, a runtime environment, and a set of key applications that can be used by developers. The platform is developed and maintained by the Open Handset Alliance [1] under Google's direction. In July 2013, Google Play, the official Google app store, has reached 50 billion downloads since its inception (INC, 2013).

The smartphone market is growing every year, according to (GARTNER, 2015) 1.2 billion smartphones were sold in 2014. Applications for these devices are extremely attractive, for example, a user can easily add new features to their device by installing the applications, which are largely free, directly from online stores. This study (IDC, 2015) shows that Android is one of the most widely used operating systems in the mobile device market.

Android applications are written in the Java language and run in their own process,

[1] http://www.openhandsetalliance.com/

within Android Runtime (ART), so applications can not access data from other applications within the device. Unlike Java Desktop applications, which run inside the Java Virtual Machine (JVM), in Android applications the connection between processes is done more decoupled. They can call any other application on the device and are available so that other applications can order them.

2.4.1 Android Abstractions

Android apps are written in the Java language. Each app runs within a separate process within the Android Runtime (ART)[2], which means that memory spaces are not shared between apps. Since it is commonplace for apps to require services provided by other apps, for example, to share a photo taken by the Camera app using Whatsapp, Android provides facilities for apps to communicate without having to share memory.

The Android application framework defines a number of basic components that are not available in general Java development (INC., 2015; CHIN et al., 2011). *Activities* interact with users by means of a graphical user interface. *Services* execute long-running operations in the background and without user interaction. *Broadcast receivers*, on the other hand, work as listeners, registering their interest in events that will be sent by the platform or other components as background broadcast messages. One important concept of the platform is the `Intent` object. It represents a message to be sent to the platform in order to request the execution of other components. It is used for communication between components, to start an activity or service asynchronously, or to deliver broadcast messages. Finally, *AsyncTasks* are employed to execute short background operations asynchronously, in order to avoid blocking the UI thread.

Intents. In Android, apps should be able to communicate while remaining isolated from a memory access and security perspective. Intents promote decoupled communication both within an app and between apps (PHILLIPS; HARDY, 2013). They can be used explicitly, in which case the intent has a specific app component as its destination, or implicitly, where the intent is sent without prior knowledge of the component that will receive it. Intents represent operations that should be executed asynchronously. At runtime, an intent is an object that is passed to a method that starts an activity or service, or sends a broadcast message. For example, to start an activity, an intent object should be passed to methods `startActivity()` or `startActivityForResult()`. The first one asynchronously

[2]ART was introduced in Android 4.4, released in September 2013, and completely replaced the Dalvik runtime environment in Android 5.0, released in June 2014.

executes some user-related functionality, such as opening a window, without producing a result for the caller. The second one indicates that a result can be produced. In this case, when the activity finishes, a call to method `onActivityResult()` occurs, delivering another intent object corresponding to the result.

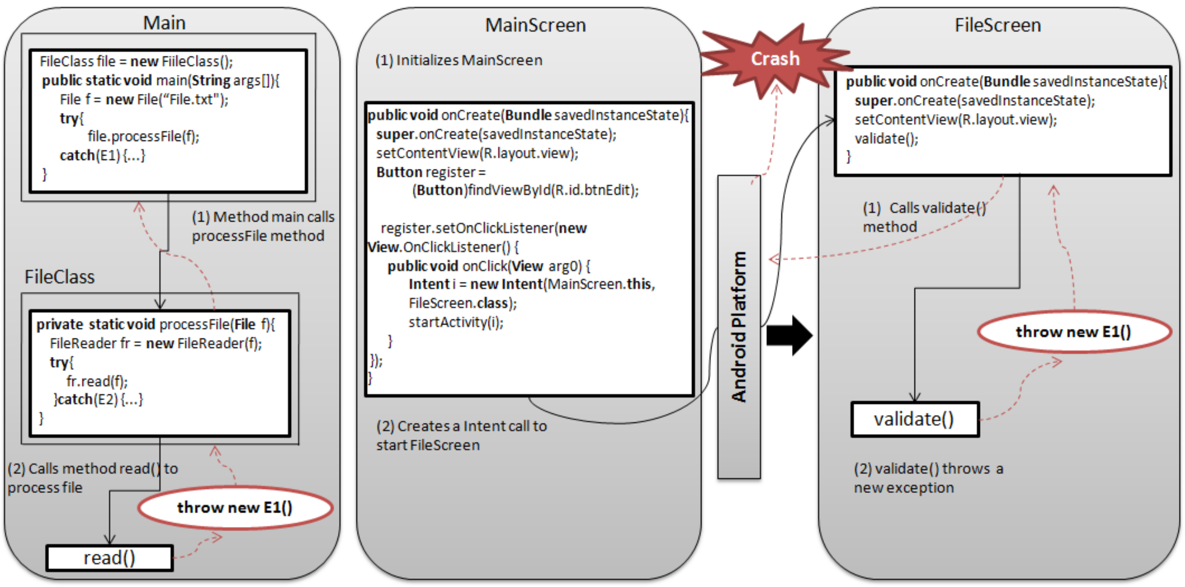
Asynctasks. In Android, short-lived (requiring a few seconds at most) blocking operations that could potentially cause the UI thread to hang should be executed within asynctasks. Asynctasks are executed asynchronously and have the goal of guaranteeing that the user interface remains responsive in spite of blocking operations such as networked communication or access to the filesystem. Asynctasks are defined by subclassing the `AsyncTask` class. Since the execution of an asynctask goes through 4 steps, the `AsyncTask` class defines 4 methods corresponding to these steps. They are invoked at runtime and can be overridden in the definition of new asynctasks. Methods `onPreExecute()`, `onProgressUpdate()`, and `onPostExecute()` are invoked by the UI thread. They are responsible for setting up the asynctask, displaying progress updates during asynctask execution, and presenting the result of the computation, respectively. Method `doInBackground()` is invoked by the background thread, i.e., is executed asynchronously, and performs the blocking operation.

2.4.2 Characterizing the Exception Handling Mechanism in Android Applications

Method calls to Android abstractions may throw exceptions. Exception handling mechanisms (AL., 2001; GOODENOUGH, 1975; PARNAS; WURGES, 1976) (EHM) are the most common approach to cope with errors in the development of software systems. In order to support the reasoning about exception flows in Android applications we present (in Figure 5) the main concepts of an exception-handling mechanism and correlate each element with the constructs available in standard Java and Android platform. An exception handling mechanism is comprised of four main concepts: the *exception*, the *exception signaler*, the *exception handler*, and the *exception model* that defines how signalers and handlers are bound (AL., 2001).

Exception Signaler. An exception is raised by an element - method, e.g., `startActivity()` - when an abnormal state is detected. In most languages an exception is usually assumed as an error, and represents an abnormal computation state (CRISTIAN, 1982; GOODENOUGH, 1975; RANDELL, 1995). Whenever an exception is raised inside an element that cannot handle it, it is signaled to the element's caller. The exception signaler is the element that detects the abnormal state and raises the exception. Figure 5 shows two hypothetical examples for Java and Android applications. In this Figure, methods `read()` and `validate()` detect an abnormal condition and raises exception E_1.

Exception Handling. The exception handler is the code invoked in response to a raised exception. It can be attached to protected regions (e.g. methods, classes and blocks of code). Handlers are responsible for performing the recovery actions necessary to bring the system back to a normal state and, whenever this is not possible, to log the exception and abort the system in an expectedly safe way.


Main
(1) Method main calls processFile method
FileClass
(2) Calls method read() to process file
throw new E1()
read()
MainScreen
(1) Initializes MainScreen
(2) Creates a Intent call to start FileScreen
Crash
Android Platform
FileScreen
(1) Calls validate() method
throw new E1()
validate()
(2) validate() throws a new exception

```
FileClass file = new FiileClass();
 public static void main(String args[]){
    File f = new File("File.txt");
    try{
        file.processFile(f);
    catch(E1) {...}
 }
```

```
private static void processFile(File f){
   FileReader fr = new FileReader(f);
   try{
      fr.read(f);
    }catch(E2) {...}
}
```

```
public void onCreate(Bundle savedInstanceState){
  super.onCreate(savedInstanceState);
  setContentView(R.layout.view);
  Button register =
          (Button)findViewById(R.id.btnEdit);

   register.setOnClickListener(new
View.OnClickListener() {
     public void onClick(View arg0) {
        Intent i = new Intent(MainScreen.this,
        FileScreen.class);
        startActivity(i);
     }
 });
}
```

```
public void onCreate(Bundle savedInstanceState){
  super.onCreate(savedInstanceState);
  setContentView(R.layout.view);
  validate();
}
```

Figure 5: Example of Exception Propagation in Java and Android applications. (OLIVEIRA et al., 2018)

Uncaught Exception. An *exception flow* is a path in a program call graph that links the signaler and the handler of an exception. For instance, in the left-hand side of Figure 5 there are two Java classes, `Main` and `FileClass`, extending `java.lang.Object`. In this scenario, the `main` method calls the `processFile` method that calls method `read` in the second step. Method `read` throws an unchecked exception `E1`[3]. If an exception is thrown in the context of the `try` block, the EHM performs at runtime the search for a proper handler. The search takes into account the list of `catch` blocks statically attached to the enclosed `try` block. The type of the exception thrown is compared with the exception types declared as arguments in each `catch` block. For the first matching pattern, the exception handler of that `catch` block is executed. Notice that in Figure 5 method `processFile` has a `try-catch` block with an exception type `E2` that does not

[3]Throughout the paper, we use the term "exception" to refer to both the object that is thrown to signal the occurrence of an error and the type defining it.

match with `E1`. If no matching pattern is found, the exception propagates up the call stack until a matching handler is found. In this scenario, the exception thrown by method `read` is caught by the `catch` block of method `main`. The exception flow of E_1 for this scenario is `read` $\rightarrow$ `processFile` $\rightarrow$ `main`. Therefore, the exception flow comprises three main moments: the exception signaling (method `read`), the exception flow through the elements of a system (method `processFile`), and the moment in which the exception is handled (method `main`). Notice that if there is no handler for a specific exception, the exception flow starts from the signaler and finishes at the program entry point. In the rest of this paper, unless it is explicitly mentioned, we use the expression *"uncaught exception"* to refer to exception flows that leave the bounds of the system (entry points) without being handled.

According to Cui and colleagues (CUI et al., 2015), one of the main differences between traditional Java programs and Android applications is that traditional Java programs use a single main method as the entry point whereas Android components can have many entry points. These entry points include Android lifecycle methods (*onCreate*, *onStart*, etc) and user-defined event handlers. These methods are invoked by the Android framework at runtime and their order of execution cannot be determined in advance (CUI et al., 2015).

For instance, in the right-hand side of Figure 5, there is an Android application with two classes `MainScreen` and `FileScreen` extending `android.app.Activity`. Initially, the Android platform creates an instance of `MainScreen` and invokes the `onCreate` method to create and register a button on the main application screen. When the user clicks the button, the `onClick` method is invoked to create an `Intent` call to start the `FileScreen` Activity. Android accepts the request and changes the current UI screen by stacking `FileScreen` on `MainScreen`, calling `FileScreen's` `onCreate` method. The `onCreate` method invokes the `validate` method that throws an unchecked exception `E1`. The EHM performs at runtime the search for a proper handler. However, this search is limited to the memory space of the current Activity. For instance, in the hypothetical scenario, exception `E1` propagates up the call stack(to method `onCreate`) until a matching handler is found. If no handler is found in the `FileScreen` Activity, the exception `E1` is propagated to the Android platform. In this scenario, the whole application will be terminated.

2.4.2.1 Use of Exception Handling in Android Applications

Java is the main language to develop Android applications. As consequence, Android applications inherit the exception handling mechanism of Java to signal and handle exceptions and that brings some problems in the use of this mechanism that have been studied in some research work recently. The study conducted by Coelho et al. (COELHO et al., 2015) analyzed 6,005 stack traces extracted from issues reported for 639 Android open source projects. They point some reasons to crashes in analysed applications: **Cross-type exception wrappings**, such as an `OutOfMemoryError` wrapped in a checked exception, undocumented runtime exceptions raised by the Android platform and third-party libraries, undocumented checked exceptions signaled by native C code.

The study by Kechagia, Maria, and Spinellis, Diomidis (KECHAGIA; SPINELLIS, 2014) examined crash stack traces from 1.800 Android applications. They were interested in finding Android API methods with **undocumented exceptions** that are part of application crashes. Their main finding is that most crashes originated from unchecked exceptions. Kechagia, Maria, and Spinellis, Diomidis (KECHAGIA et al., 2018) examine the use of the Java exception types in the Android platform's Application Programming Interface (API) reference documentation and their impact on the stability of Android applications. They discover that almost 10% of the undocumented exceptions that static analysis can find in the Android platform's API source code manifest themselves in crashes. Additionally, they observe that 38% of the undocumented exceptions that developers use in their client applications to handle API methods also manifest themselves in crashes.

The work by Oliveira, Juliana, et al. (OLIVEIRA et al., 2016) shows some limitations that can affect negatively the robustness of Android applications. The findings shows that there are an **excessive use of unchecked exceptions** by developers and Android platform. These unchecked exceptions were signaled mainly by libraries reused by Android and by methods defined in the Android platform. The study points there are an **ineffective use of exception handlers** by developers that can be related to the excessive use of unchecked exceptions and **outdated exception interfaces** that affect program comprehension, since it is not possible to know the exceptions that a method may throw just by looking at its exception interface.

2.5 Static Analysis

Static analysis examines the source code of a compile-time program in order to point out reasons for the occurrence of all possible behaviors that may arise at runtime. According to Landi, William (LANDI, 1992) the static analysis extracts semantic information about the program. It provides assurances that the results of the analysis will be an accurate description of the program's behavior, regardless of the type of input it will receive or what environment it is running (ERNST, 2003).

To perform an analysis of a particular program the static analysis engine constructs a model of the program state and then determines how it will react to that state. The analysis should maintain and control the different types of states that may occur at each program execution. In practice, it is cumbersome to consider all possible program states at runtime, one program can receive many inputs from different users. Therefore, static analyzes define an abstract model of the program state. With this strategy there is a lot of information loss, however, this model is easier to manipulate than a model with more fidelity about the execution states of the program.

One of the major drawbacks of using static analysis is that for large programs, executions tend to have a long waiting time due to the amount of computation performed. Article by Johnson, Brittany, et al. (JOHNSON et al., 2013) carried out a study with software developers in order to evaluate the use of static analysis tools. This study points to some reasons why these developers do not use these tools to find bugs.

Among these reasons is the fact that the tools do not present the results of the analysis in an intuitive way so that the developer can understand what the result means and what measure he must take in front of it. This is related to the production of many false positive and the large number of warnings generated by such tools, problems already known in the use of static analysis (SHEN; FANG; ZHAO, 2011) (AYEWAH et al., 2008). Despite these drawbacks, the use of static analysis, especially in the context of exception handling, yields useful results that can contribute significantly to improving system robustness and quality. With the use of static analysis it is possible to capture relevant information about the exceptional flow of an application.

Since most modern programming languages provide exception handling mechanisms with constructs that allow the throwing of exceptions and delimiting a block of code to handle such exceptions. Obtaining this information can be done by analyzing the source code of the applications without the need to execute them, which is very advantageous in the context of mobile applications. Because the analysis can be done without the need to configure and deploy the application on the mobile device.

2.6 EFlow Model

EFlow model (CACHO et al., 2008; CACHO; COTTENIER; GARCIA, 2008) is grounded on mechanisms to explicitly represent global exceptional behavior. This specification enables software developers to establish constraints governing the implementation of non-local exceptions flows. An exception handling specification is composed of two abstractions: `explicit exception channel` and `pluggable handlers`.

An explicit exception channel is an abstract duct through which exceptions flow from a raising site to a handling site. More precisely, an explicit exception channel (EEC) is a 5-tuple consisting of: (1) a set of exception types E, (2) a set of raising sites RS; (3) a set of handling sites HS; (4) a set of intermediate sites IS; and (5) a function EI that specifies the channel's exception interface.

Exception types, as the name indicates, are types that, at runtime, are instantiated to exceptions that flow through the channel. The raising sites are loci of computation where exceptions from E can be raised. The actual erroneous condition that must be detected to raise an exception depends on the semantics of the application and on the assumed failure model. For reasoning about exception flow, the fault that caused an exception to be raised is not important, just the fact that the exception was raised. The handling sites of an explicit exception channel are loci of computation where exceptions from E are handled, potentially being re-raised or resulting in the raising of new exceptions. In languages such as Java, both raising and handling sites are methods, the program elements that throw and handle exceptions.

If an explicit exception channel has no associated handlers for one or more of the exceptions that flow through it, it is necessary to define its exception interface. The latter is a statically verifiable list of exceptions that a channel signals to its enclosing context, similarly to Java's throws clause. In the model, the exception interface is defined as a function (Ex1αEx2) that translates exceptions flowing (Ex1) through the channel to exceptions signaled (Ex2) to the enclosing EHC.

Raising and handling sites are the two ends of an explicit exception channel. Handling sites can be potentially any node in the method call graph that results from concatenating all maximal chains of method invocations starting in elements from HS and ending in elements from RS. All the nodes in such graph that are neither handling nor raising sites are considered intermediate sites. Intermediate sites comprise the loci of computation through which an exception passes from the raising site on its way to the handling site. Intermediate sites in Java are methods that indicate in their interfaces the exceptions that they throw, i.e. exceptions are just propagated through them, without side effects to program behavior. Note that the notions of handling, raising, and intermediate site are purely conceptual and depend on the specification of the explicit exception channel. They are also inherently recursive. For example, an intermediate site of an explicit exception channel can be considered the raising site of another channel.

A pluggable handler is an exception handler that can be associated to arbitrary EHCs, thus separating error handling code from normal code. A single pluggable handler can be associated, for example, to a method call in a class C1, two different method declarations in another class, C2, and all methods in a third class C3. In this sense, they are an improvement over traditional notions of exception handler. Another difference is that a pluggable handler exists independently of the EHCs to which it is associated. Therefore, these handlers can be reused both within an application and across different applications.

2.6.1 EFlow Implementations

The first two implementations of the Eflow model were: EJFlow (CACHO et al., 2008) and ESFlow (CACHO; COTTENIER; GARCIA, 2008). The EJFlow implementation extends the AspectJ programming language with the aim of promoting enhanced modularity, reliability, and maintainability of exception handling. The goal of the second implementation, called ESFlow, is to demonstrate the wide applicability and generality of the EFlow model. ESFlow extends the aspect-oriented language constructs and the control-flow analysis of the Motorola WEAVR (COTTENIER; BERG; ELRAD, 2007a, 2007b) with the aim of promoting platform-independent representation of global exceptional behavior. ESFlow allows a modular and maintainable design of exception handling based on executable statecharts (HAREL, 1987). The other difference between EJFlow and ESFlow is that the former relies on synchronous communication between modules, while the latter is based on asynchronous communication.

EJCFlow (ARAÚJO et al., 2012), the third implementation of EFlow model, is a JCML extension that implements the EFlow model to support Java Card applications. EJCFlow provides means for developers to define explicit exception channels and pluggable handlers in terms of the abstractions supported by JCML. This implementation is specific for Java

Card applications to ensure that an exception raised inside the card will appear for the developer of the application with a proper information of the error, without EJCFlow the only information that the developer has is an error code.

Another implementation of the EFlow model is called ECSFlow (FILHO, 2016). This implementation was developed for C# applications to provide an exception handling mechanism that favors the reliability of systems, satisfying characteristics such as robustness and maintainability. The implementation uses elements of the language to represent the EFlow abstractions and is provided for the developer as an Add-in to facilitate the integration with the Visual Studio tool.

In this work we propose DroidEH that implements the EFlow model as well. Unlike previous implementations, DroidEH is focused on mobile Android applications that have a different architecture and specific exception handling limitations. Also DroidEH inherents the concepts of Holistic Fault Tolerance (GENSH et al., 2017a) to enhance the EFlow model.

2.7 Holistic Fault Tolerance

Holistic fault tolerance has been recently proposed (GENSH; ROMANOVSKY; YAKOVLEV, 2016; GENSH et al., 2017a) to support engineering of a cross-layer fault tolerance coordination to impose modularity to perform cross-cutting error detection and recovery for mobile many-core applications and to ensure improved system performance and power consumption. In the core of this approach is the architectural pattern (GENSH et al., 2017a) and modelling techniques (GENSH et al., 2017b) which allow the developers to make optimal decisions about holistic involvement of system components in system level error detection and error recovery for each specific error.

Figure 6 shows the HFT architecture blueprint (GENSH et al., 2017a). The architecture assumes that the application is build out of components (C1-C7) whose responsibility is to deliver the main system functionality. The core of this architecture is a special component called the HFT controller, which is supported by several HFT agents. These elements of the architecture ensure dependable and optimal system operation. In addition, they provide a clear view of the system FT mechanisms. The HFT controller coordinates system-wide FT with the assistance of the HFT agents that simplify the implementation and improve the scalability of the HFT controller. Each HFT agent acts as an intermediary between the HFT controller and one or several system components. It is possible to go without the HFT agents for small applications, given that the corresponding functionality

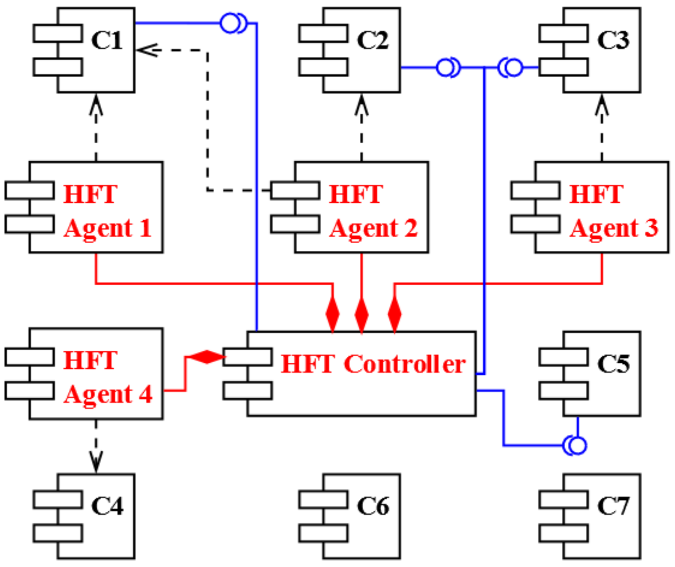


Figure 6: HFT architecture from (GENSH et al., 2017a).

will be implemented by the HFT controller.

The FT mechanisms in the given architecture are distributed across the entire system, but coordinated centrally by the HFT controller. In some cases, it is worth to introduce some redundancy in FT mechanisms in such a way that the same error could be handled by the component itself and by the HFT agent. The decision on suitable error handling scenarios will be made by the HFT controller depending on the current system state. Such an approach provides the flexibility in the choice of the optimal error recovery scenario.

A system designer should choose which system components will participate or will be included in the HFT behaviour and which components will just provide their functionality without being affected by the HFT controller and agents.

As part of the HFT architecture, (GENSH et al., 2017a) proposes to apply HFT for modes and mode control, so that each mode supports a chosen interplay between reliability, performance and energy consumption. In this context, an Operation Mode (OM) could be defined as a functional state of the system. Another application of OMs is to support system degradation by providing a non-functional distinction when one mode describes full functionality of the system, whereas another mode is used for dealing with exceptional conditions, including recovery and reconfiguration with degradation.

3 Investigating the Impact of Exception Handling on the Robustness and Energy Consumption of Android Applications

This chapter shows in details, two studies carried out to investigate the impact of exception handling on robustness and energy consumption of Android applications. These two non-functional requirements need to be taken into account when developing Android applications, once they may impact negatively on the user experience and quality of the applications. Android applications have different characteristics such as, different abstractions, multiple entry points, etc. These abstractions are used intensively during the development of applications, so the first study focuses on analyzing the relationship between the usage of Android abstractions and uncaught exceptions, to evaluate the impact of exception handling on the robustness. It has analyzed 112 versions extracted from 16 software projects covering a number of domains. The second study was carried out with a different focus, investigating the impact of the use of exception handling strategies on the energy consumption of Android applications. It has analyzed 10 Android applications.

3.1 A Maintenance-Centric Study on the Relationship Between Android Abstractions and Uncaught Exceptions

A previous work by Oliveira, Juliana, et al. (OLIVEIRA et al., 2016) analyzed Java standard applications and Android applications during the evolution and showed that there is an excessive use of unchecked exceptions by developers and Android platform compared to Java standard applications. This study points out that there is an ineffective use of exception handlers by developers that can be related to the excessive use of unchecked exceptions and outdated exception interfaces that affect program comprehension, since it is not possible to know the exceptions that a method may throw just by looking at its exception interface. The study presented in this chapter continues this previous work focusing on Android applications. The work by Oliveira, Juliana, et al. (OLIVEIRA et al., 2016) shows there are general problems related to the use of Java EHM in Android applications that can affect the robustness. Since Android applications have different characteristics such as, different abstractions, multiple entry points and problems like the *"Terminate ALL"* approach, this study deepens the analysis of the use of Android-specific abstractions and their impact on robustness of the Android applications during evolution. This analysis is important to development of the exception handling mechanism proposed in this research work and explained in section 4.2 of chapter 4, because this mechanism needs to take into account how the use of these Android abstractions impact robustness, so that, this impact is lessened by the use of the mechanism.

All the Android-specific abstractions can throw exceptions, with different effects. For example, an `ActivityNotFoundException` thrown in a call to the `startActivity` method can be caught similarly to any Java exception. However, if an exception is thrown during the execution of a started activity, the only way of handling it is by creating an instance of `UncaughtExceptionHandler` and associating it with the GUI thread. Otherwise, these exceptions will always cause the application to crash ("Force Close" in Android terminology). Both asynctasks and intent-related calls can end their execution raising an exception. Since they execute asynchronously, the Android API provides means for developers to check if execution finished with an exception and, if so, to throw that exception in the UI thread, where it can be handled. On the other hand, asynchronous execution of operations means that some exceptions might end up causing threads to fail.

Android's abstractions are intensively used. In the group of 16 Android apps we have analyzed, there is an average of approximately 53 activity-related calls and 9.3 asynctask-related calls per version of each app. This means that, on the average, one in every 500 lines of code (LOC) performs an intent-related call and one in every 2880 uses an asynctask.To get an intuition of how big these numbers are, we can examine the usage of certain constructs in non-Android Java applications. For example, previous work (PINTO et al., 2015) has shown that, in a population of 1723 stable and mature Java (non-Android) applications from SourceForge, `synchronized` blocks, one of the most common constructs for concurrency control in the language, appear on the average once for every 1517 LOC[1].

[1]The paper reports an average of 65.93 `synchronized` blocks per 100KLOC. This average was

Invocations of the `notifyAll` method, one of Java's basic building blocks to implement thread synchronization, appear on the average once for every 3667.62 LOC. In spite of the intensive use of Android abstractions and the potential for uncaught exceptions stemming from their use, no previous study has analyzed the relationship between these two factors. Understanding if and when Android abstractions cause programs to fail, in particular due to uncaught exceptions, can make developers use them in a more disciplined way. Testers can also leverage this knowledge to create specific tests for common error conditions related to these abstractions. At the same time, bug finding tools can be augmented to look for unprotected uses of these abstractions, e.g., calls to the `get` method of class `AsyncTask` for which there is no handler and activities that run in a GUI thread that has no associated `UncaughtExceptionHandler`.

This section presents an empirical study to analyze the relationship between the usage of Android abstractions and uncaught exceptions. Our approach is quantitative and maintenance-centric. We analyzed changes to both normal and exception handling code (or "exceptional code", the `catch` blocks) in 112 versions extracted from 16 software projects covering a number of domains, amounting to more than 3 million LOC. Change impact analysis and exception flow analysis[2] (SCHAEFER; BUNDY, 1993; ROBILLARD; MURPHY, 2003) were performed on those versions of the projects. Since developers often report errors stemming from uncaught exceptions (EBERT; CASTOR; SEREBRENIK, 2015; SENA et al., 2016), similarly to previous work (CABRAL; MARQUES, 2007; COELHO et al., 2008; ROBILLARD; MURPHY, 2003) we consider the presence of uncaught exception flows to be an indicator of lack of robustness.

Next section presents the experimental procedure, including the hypothesis of the study, the sample of applications and variables and metrics. Section 3.1.2 describes the results found in the study with the analysis of the results in the context of Android abstractions and changes to exception handling code and Android abstractions and robustness. Finally, section 3.1.3 presents the threats to validity.

3.1.1 Experimental Procedures

As described in the previous section, the use of Android's abstractions can lead to a number of different exceptions. These exceptions may cause programs to fail if not

obtained from 612,897,893 LOC, which have 404,019 `synchronized` blocks. This yields an average of one `synchronized` block for every 1517 LOC.

[2]An exception flow is a path in the program call graph that may be traversed by an exception that it throws. If an exception flow is uncaught, the corresponding path reaches one of the program entry points.

properly handled. However, there is limited empirical knowledge about the impact of using Android's abstractions on program robustness, in general, and on uncaught exceptions, in particular. Therefore, the goal of this empirical study is to investigate if and to what extent typical uses of these abstractions affect software robustness during software evolution as a consequence of uncaught exceptions.

We describe the experimental procedure of this study in four parts: (i) definition of the the study hypotheses (Section 3.1.1.1); (ii) selection of the sample of subject applications (Section 3.1.1.2); (iii) definition of the variables of the study and the suite of metrics to be computed from the source code and binaries of the subject applications (Section 3.1.1.3); and (iv) finally, statistical analysis of the collected metrics using the statistical package SPSS (Section 3.1.2).

3.1.1.1 Hypotheses

Android development introduces a number of new abstractions that are not part of traditional Java development. Some of them are executed asynchronously or using a callback style of programming. Usage of all of these abstractions may throw exceptions that can cause apps to crash(WU et al., 2017). In addition, most of these exceptions are unchecked and the Java compiler does not require them to be handled by the apps(COELHO et al., 2015). This is in contrast to how Java frameworks are typically built. The latter usually employ mostly checked exceptions (BARBOSA; GARCIA, 2011; NAKSHATRI; HEGDE; THANDRA, 2016) since checked exceptions are a form of compiler-enforced documentation. In addition, considering a more general perspective, although there is a considerable body of knowledge on how exceptions are thrown, propagated, and handled in Java programs (CABRAL; MARQUES, 2007; FU; RYDER, 2007; ROBILLARD; MURPHY, 2000, 2003; NAKSHATRI; HEGDE; THANDRA, 2016; KERY; GOUES; MYERS, 2016), the same cannot be said about Android applications, although this is starting to change (COELHO et al., 2015; KECHAGIA; SPINELLIS, 2014; WU et al., 2017). These factors motivate a more in-depth analysis about the relationship between Android abstractions and exception handling. This study relies on the analysis of the following hypotheses, which are set up as null hypotheses:

- **Hypothesis 1**: There is no significant relationship between changes in the number of method calls to Android abstractions and changes in the amount of exception handling code (code within `catch` blocks).

- **Hypothesis 2**: There is no significant relationship between changes in the number of classes that extend Android abstractions and changes in the amount of exception handling code (code within `catch` blocks).
- **Hypothesis 3**: There is no significant relationship between changes in the number of method calls to Android abstractions and the number of uncaught exception flows in a system.
- **Hypothesis 4**: There is no significant relationship between changes in the number of classes that extend Android abstractions and the number of uncaught exception flows in a system.

The first two hypotheses aim to investigate whether changes in the number of uses of Android abstractions (methods and classes) are associated to changes in the amount of exception handling code. Hypotheses 3 and 4 investigate whether variations in the number of uses of these abstractions have side effects in terms of uncaught exception flows.

3.1.1.2 Sample

We selected our sample of subject applications based on the categories provided by Google Play. We divided our subject applications in four categories: (i) **Social Network**: Apps that connect people by means of text, voice, photos, or video; (ii)**Communication**: Apps responsible for using 3G or WiFi to message with friends and family; (iii) **Miscellaneous**: Apps that provide browser features, ebook reader, and so on; (iv) **News Reader/Media**: Apps that provide information about current events. In our sample, we include apps with a high number of installations(more than 50k) and with more than 7 versions available to download or with public version control systems available. In addition to these requirements, for us to be able to perform the manual and static analyses, applications need to provide the source code of each version and the APK file.

In the process of selecting the subject applications, we identified that many of them did not have significant changes between some pairs of versions. This is the reason why we cannot choose the versions to be analyzed solely based on when they were released. In these cases, we chose applications with a longer development time, and skipped minor versions, e.g., analyzed changes between versions 1.1 and 1.3, instead of 1.1 and 1.2. Only apps with at least 7 versions available were considered in the study. Our final sample comprised a set of 16 open source Android apps and is particularly diverse in the way exception handling is employed. For instance, when we consider a classification of how exception handling code can be structured (Castor Filho; GARCIA; RUBIRA, 2007), we could find almost all the categories of the aforementioned classification among the 16 subject applications, including nested exception handlers, masking handlers, context-dependent handlers, and context-affecting handlers. We could also observe that the behavior of exception handlers significantly varied in terms of their purpose (CABRAL; MARQUES, 2007), ranging from error logging to application-specific recovery actions.

Table 1: Subject programs of the study

	Name	Versions	Pairs	AverageLOC	Android API
Social Network	TweetLanes	1.2.1-release; 1.3.0beta1; 1.3.0beta4; 1.3.0release; 1.4.0beta1; 1.4.0beta2; 1.4.0release;	6	26.022,33	17
	Twidere	0.2.9.7update1; 0.2.9.8; 0.2.9.8update2; 0.2.9.9; 0.2.9.10; 0.2.9.11; 0.2.9.12;	6	75.765,66	18, 19
	Impeller	0.7.0; 0.8.2; 0.8.5; 0.8.6; 0.9.5; 0.9.5a; 0.9.5b;	6	8.930,66	18, 19
	Andstatus	6.3; 7.0; 7.1; 8.0; 8.1; 8.2; 8.4;	6	20.130,50	10
	SubTotal		**24**	**785.095,00**	
Communication	ChatSecure	12.7.1; 13.0.3; 13.0.4; 13.0.6; 13.0.9; 13.2.0alpha6; 13.2.0beta1;	6	34.522,00	18, 19
	ConnectBot	1.1; 1.3; 1.5.4; 1.6.1; 1.6.2; 1.7.1; 2014-03-02-12-38-08;	6	31.666,16	3, 6, 8, 15
	TextSecure	2.0.4; 2.0.5; 2.0.6; 2.0.7; 2.0.8; 2.1.2; 2.1.6;	6	46.521,50	19
	Adblock Plus	1.0; 1.0.1; 1.1; 1.1.1; 1.1.4; 1.2; 1.2.1;	6	9.872,66	7, 16
	SubTotal		**24**	**735.494,00**	
Miscellaneous	K9mail	4.700; 4.701; 4.800; 4.801; 4.802; 4.803; 4.804;	6	77.093,66	17, 19
	TintBrowser	1.3; 1.4; 1.5; 1.6; 1.6.1; 1.7; 1.8;	6	11.347,83	14, 17
	FBReader	1.10.0.4; 1.10.2; 1.10.3.2; 2.0.4; 2.0.5; 2.0.5.2; 2.0.6;	6	64.183,66	14
	Cgeo	20140330; 20140331; 20140401; 20140410-legacy; 20140419; 20140430; 20140514;	6	49.783,33	8, 9
	SubTotal		**24**	**1.214.451,00**	
News Reader / Media	Serenity for Android	1.5.2; 1.5.3; 1.5.4; 1.5.5; 1.6.1; 1.7.3; 1.7.4a ;	6	32616,83	13, 19
	RedReader	1.6.5; 1.6.9; 1.8.0; 1.8.2; 1.8.3; 1.8.5.2; 1.8.6.3;	6	15592,16	16
	Vanilla	0.9.17; 0.9.18; 0.9.20; 0.9.21; 0.9.22; 0.9.23; 0.9.24;	6	10495	16
	VLC for Android	0.9.0; 0.9.2; 0.9.4; 0.9.6; 0.9.7.1; 0.9.9; 1.0.0;	6	27355,83	19, 21
	SubTotal		**24**	**516.359,00**	
Total			**96**	**3.251.399,00**	

Table 1 reports the name, number of analyzed version pairs, average number of lines of code and the version of Android platform's API of each subject applications. For the TintBrowser app, for instance, we analyzed 6 pairs of versions: 1.3-1.4, 1.4-1.5, 1.5-1.6, 1.6-1.6.1, 1.6.1-1.7 and 1.7-1.8. The last column shows the version of Android platform's API used in each version of the subject applications. Many of them used more than one version of the API throughout its evolution. For example, the Twidere application started using version 19 of the API when it reached version 0.2.9.9 and employed version 18 prior to that. For each category, Table 1 also shows the total number of pairs and the total number of LOC analyzed. The last row shows the total number of pairs and the total number of LOC analyzed in this study.

3.1.1.3 Variables and Metrics

The selected variables are metrics quantifying characteristics of both normal and exceptional code. We chose different metrics to capture changes in the normal and the exceptional code during software evolution, and also metrics that quantify the robustness of a program. Therefore, the variables of interest of this study encompass a suite of metrics classified in three categories: size metrics, robustness metrics, and change metrics. The following subsections describe each suite of metrics and how they were computed.

Robustness Metrics: Robustness is the ability of a program to properly cope with errors during its execution (LEE; ANDERSON, 1990). When an exception is thrown, if a corresponding handler cannot be found, program robustness is decreased. Other factors can also influence robustness, for example, security vulnerabilities, excessive resource consumption, and race conditions. These problems, albeit very important, usually do not directly result in exceptions being thrown at runtime. Thus, we do not study them in this work.

In order to measure software robustness, we followed the approach adopted by previous work(CACHO et al., 2008; COELHO et al., 2008; CACHO et al., 2014, 2014). We used exception flow information as an indicative of robustness (ROBILLARD; MURPHY, 2003; FU; RYDER, 2007; COELHO et al., 2008; MAJI et al., 2012). Exception flow is a path in a program call graph that links the method where the exception is thrown to the method where the exception is handled. If there is no handler for a specific exception, the exception flow starts in the method where the exception was thrown and finishes at an entry point.

In standard Java applications, the entry point is the `main` method if the exception was thrown in the main thread, or the `run` method in case it was thrown in a separate thread. Unlike standard Java programs, Android applications may have many entry points. These entry points include Android lifecycle methods, like `onCreate` method, and user-defined event handlers, like the `onClick` method that handles a button click performed by the user. These methods are invoked by the Android framework at runtime and their order of execution cannot be determined in advance. In this work we followed the approach adopted by (CUI et al., 2015) to identify entry points for each component of an Android application. In the context of our analysis of program robustness, we used the *Uncaught Exception Flow* (or *Uncaught Flow*) metric. The uncaught exception flow is often used as an indicative of software robustness (ROBILLARD; MURPHY, 2003; FU; RYDER, 2007; COELHO et al., 2008; MAJI et al., 2012). *Uncaught Flow* counts the number of exception flows that reach an entry point (`main` or `run` method) where the corresponding exception is not caught. This is an indicator of a potential fault in exception handling behavior. Uncaught exceptions terminate the execution of a program or thread and, since exceptions indicate errors, an uncaught exception is synonym with an error the program failed to handle.

We employed an extended version of the eFlowMining (GARCIA; CACHO, 2011) tool to collect the robustness metric. eFlowMining is a multi-language static analysis tool that uses the approach proposed by (FU; RYDER, 2007) to perform an inter-procedural and intra-procedural dataflow analysis. This extended version of eFlowMining uses the Dexpler software package (BARTEL et al., 2012) to collect the structure (call graph, methods, exceptions, etc) of Android applications. Then, the tool generates the exception flows and computes the *Uncaught Flow* metric for all exceptions, explicitly thrown by the application or explicitly thrown by library methods. In this study we are assuming that only one exception is thrown at a time — the same assumption considered in (FU; RYDER, 2007). Unlike (CABRAL; MARQUES, 2007; CACHO et al., 2014), we have taken into account all exception flow types, including ones originating from unchecked exceptions, since recent work (KECHAGIA; SPINELLIS, 2014; FRASER; ARCURI, 2015; KIM et al., 2013; COELHO et al., 2015) has found that `NullPointerException`, `IllegalArgumentException`, and `IllegalStateException` are among the most common reasons why Android applications crash.

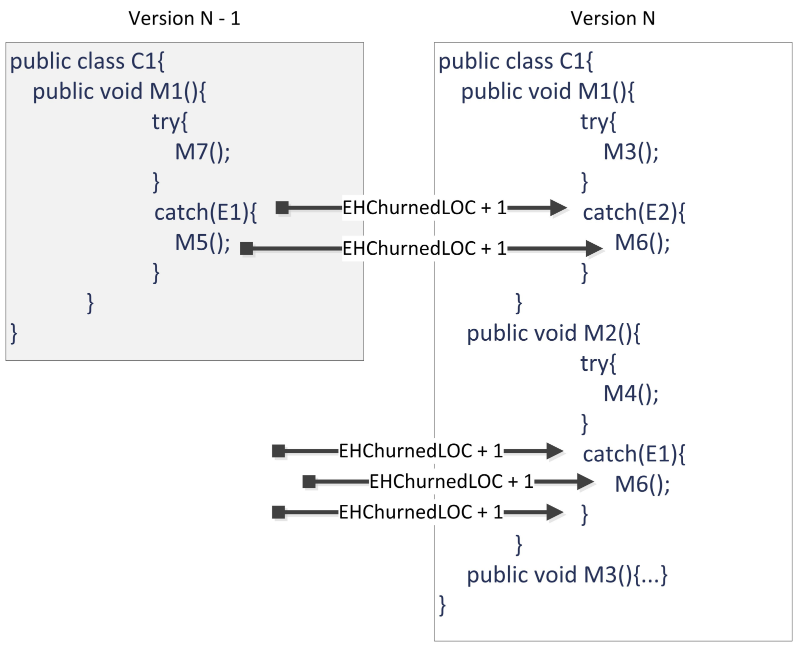


Figure 7: Collecting change impact measure: An example

Change Metrics: In order to quantify the changes in exception handling code, we collected the *EHChurnedLOC* change impact metric (YAU; COLLOFELLO, 1985). This change impact metric is computed based on the difference between the source code of a baseline version and the source code of a subsequent version. *EHChurnedLOC* counts the number of lines of exception handling code added and changed between a pair of versions. We considered to be exception handling code only the code that comprises the `catch` blocks. In the example presented in Figure 7, two lines of exception handling code are changed (within the first catch block) and three lines are added. Hence, the value of *EHChurnedLOC* is 5.

Our main goal during the change impact analysis was to gather deeper knowledge about the recurring change scenarios observed and their impact on software robustness. Therefore, we had to not only compute the change metrics, but also describe each observed change scenario and assess its impact on software robustness. Since part of this information is inherently qualitative, we could not rely on any existing static analysis tool to automatically extract it from the source code, nor implement a tool of our own. In this manner, we had to perform a manual inspection in the source code to textually describe the change scenarios and assess the impact of the observed changes on software robustness. These tasks were performed by a group of three master's students and two researchers who acted as reviewers. These researchers are specialists on error handling and have multiple papers published on the subject (Castor Filho et al., 2006; CACHO et al., 2014, 2014, 2008; EBERT; CASTOR; SEREBRENIK, 2015; OLIVEIRA et al., 2016). Each master's student performed a manual inspection on the source code of the versions of a given subject applications and simultaneously: (i) computed the change metrics, (ii) textually described the observed change scenarios and (iii) assessed the impact of the changes on software robustness by means of an exception flow analysis. The data produced by the master's students was reviewed by the two researchers. Further clarifications, modification, or improvements were performed by the master's students when divergences were identified by the reviewers. Whenever necessary, the students performed an open discussion with the reviewers to resolve conflicts and reach consensus on the data produced. If, on the one hand, this manual inspection process did not allow us to scale our study to a larger sample, on the other hand it provided us with more accurate data and a better understanding of the observed change scenarios. Moreover, it also allowed us to identify and categorize change scenarios that improved or deteriorated the robustness of the subject applications. Next, each task is described in more detail.

Change metrics computation and textual description. The change impact metrics were manually computed with the aid of the DiffMerge tool [3]. DiffMerge performs a

[3]DiffMerge page: http://www.sourcegear.com/diffmerge/

Table 2: Change Scenarios and Corresponding Descriptions

	Scenarios	Description
Changes to Exceptional Code	Generic Handler added	A catch block which catches a generic exception is added
	Empty Generic Handler added	An empty catch block is added to handle a generic exception
	Generic Handler added to rethrow exception	A catch block which catches a generic exception is added to construct a new exception which is thrown
	Generic Handler removed	A catch block which catches a generic exception is removed
	Specialized Handler added	A catch block which catches a specific exception is added
	Empty Specialized Handler added	An empty catch blocks is added to handle a specific exception
	Specialized Handler added to rethrow exception	A specific handler is added to construct a new exception which is thrown
	Specialized Handler removed	A specialized handler is removed
	Changing the Exception Handling Policy	Changes how a specific exception type is handled
	Changing the catch block to use normal code	The catch block needs to change its code in order to use some references of the normal code
	Only modified the Exception Interface	Only the exception interface is added, removed or modified
	No changes to catch blocks	The try blocks changes but the catch block is not modified.
Changes to Normal Code	Try block added to existing method	A try block is added to an existing method declaration
	New method added with try block	A new method declaration with a try block is added
	New method invocation added	A new method invocation is added within a try bock
	Variables modified	A variable within a try block is modified
	Method call replaced	Replacing the method call by another
	Intent call added	An Intent call is added to the method
	Control flow modified	A control flow statement is modified within a try block
	Try block removed	A try block is removed from a method declaration
	Method with try block removed	A method declaration with a try block is removed from the project
	No changes to try blocks	The catch block changes but the try block is not modified.

visual side-by-side comparison of two folders, showing which files are present in only one folder, as well as file pairs (two files with the same name but in different folders) that are identical or different. For file pairs with differences, the tool also graphically highlights the changes between the two files. The DiffMerge tool was used to compare the project folder of two subsequent versions of the subject applications. For each file pair with differences, the master students manually computed the change impact metrics by inspecting the differences pinpointed by the tool. For each change scenario analyzed, the students also textually described the observed changes that occurred within `try` blocks, `catch` blocks, and exception interface. For instance, in one of the change scenarios we analyzed, one of the students described the following: “We observed a new IF statement added to the try block and a new method invocation added to the catch block”. On a further step, the researchers applied a coding technique to the textual descriptions of each change scenario in order to extract categories of change scenarios, divided in categories of changes in the normal code and categories of changes in the exceptional code. Table 2 presents the result

Table 3: Occurrence of Change Scenarios per Project

	Application Name															
Change Scenarios	AdBlockPlus Android	AndStatus	cgeo	ChatSecure Android	ConnectBot	FBReaderJ	Impeller	K-9	RedReader	Serenity	TextSecure	TintBrowser	TweetLanes	twidere	vanilla	VLCAndroid
Try block added to existing method	3	5	9	19	30	6	7	1	14	5	7	1	2	10	0	6
New method with try block added	2	13	35	26	22	18	10	2	5	1	13	5	7	24	0	1
Try block removed	5	18	7	12	9	8	3	0	2	0	4	2	0	4	1	0
Method with try block removed	9	4	36	10	8	13	8	1	1	0	21	1	0	7	0	5
Method call added	2	0	14	15	7	2	2	2	1	0	10	0	2	1	1	1
Variables modified	1	16	1	5	25	0	4	0	3	0	0	0	1	28	1	14
Control flow modified	2	20	6	20	14	0	5	1	6	0	9	2	8	8	2	4
Method call replaced	0	10	20	11	4	6	0	9	1	0	1	0	0	10	0	0
Intent call added	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
No changes to try block	3	7	0	4	4	2	0	0	3	0	14	1	3	2	0	2
No changes to catch block	9	34	45	43	49	16	10	16	11	0	28	4	11	44	3	19
Specialized Handler added	3	4	8	30	34	8	9	2	4	5	21	6	11	16	0	4
Generic Handler added	3	7	23	9	12	2	3	1	3	1	4	0	0	2	0	2
Empty Generic Handler added	0	0	3	4	0	2	0	0	0	0	1	0	1	1	0	1
Generic Handler added to rethrow exception	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0
Specialized Handler added to rethrow exception	0	4	0	0	3	1	0	0	11	0	1	0	0	2	0	0
Empty Specialized Handler added	0	0	4	5	2	4	1	0	1	0	0	1	0	4	0	0
Generic Handler removed	2	8	22	8	6	11	0	0	0	0	1	2	0	0	1	0
Specialized Handler removed	12	12	20	15	11	7	11	0	3	0	20	1	0	14	0	5
Changing the Exception Handling Policy	0	8	1	12	8	2	2	0	2	0	4	0	1	2	1	2
Changing the catch block to use normal code	0	15	0	0	3	0	0	0	0	0	0	0	0	0	0	1
Only modified the Exception Interface	1	12	0	1	30	7	2	0	0	0	24	0	2	32	0	0

of this categorization.

Table 3 shows the occurrence of the change scenarios in the normal code and the exceptional code (Table 2), throughout all the versions of the analysed applications. It is possible to see that the *No changes to catch block* scenario occurs in all applications, most often in the *ConnectBot* application. The occurrence of the *No changes in the try block* occurs with a lower frequency and in some applications it did not occur. These results indicate that, in general, there were more changes in the normal code than in in the exceptional code. We also noticed that 9 out of 16 applications had changes related to the exceptional interface of the methods.

Exception flow analysis. Simultaneously to the computation of the change metrics, the students also performed exception flow analysis for each identified change scenario. The exception flow analysis was performed with the aid of the eFlowMining tool. For each pair of subsequent versions analyzed, eFlowMining computed the difference between the number of uncaught exception flows observed for each method. The difference was computed as: $\Delta Uncaught = Uncaught_{Subsequent} - Uncaught_{Baseline}$.If the value of the difference was higher than zero, then it meant that the number of uncaught exception flows increased during the evolution. On the other hand, if the difference was lower than zero, then it meant that the number of uncaught exception flows decreased during the evolution, thus suggesting that the occurrence of uncaught exceptions became less likely. Finally, if the value of the difference was equal to zero, then it meant that no changes were observed in the number of uncaught exception flows; therefore, the change scenarios observed had no impact on software robustness.

In this manner, by combining the change metrics values, the categories of change scenarios extracted from the textual descriptions, and the exception flow analysis, we were able to better understand which categories of change scenarios actually improved or deteriorated the robustness of the subject applications. In particular, the manual inspection process allowed us to systematically discover the scenarios that were more prone to generate uncaught exception flows.

Table 4: List of methods calls analyzed

		Methods	**List of Thrown Exception**
Type of Android Call	Activity	startActivity; startActivities; startActivityForResult; startActivityFromFragment; startActivityFromChild; startActivityIfNeeded	android.content.Activity NotFoundException; java.lang.IllegalState Exception; java.lang.IllegalArgument Exception
	Service	startService; bindService; unbindService	java.lang.SecurityException
	AsyncTask	execute; executeOnExecutor	java.lang.IllegalStateException
	BroadCast	sendBroadcast; sendOrderedBroadcast; sendBroadcastAsUser; sendOrderedBroadcastAsUser	-

Size Metrics: Size metrics capture the basic structure of the normal and the exception handling code of a software system. The size metrics used in this study are the following. *NClasses* counts the number of classes of each version of a system that extend `android.app.Activity`, `android.app.Service`, `android.os.AsyncTask`, or `android.content.BroadcastReceiver`. *NMethodCall* counts the number of method calls related to activities, services, asynctasks, and broadcasts, respectively, for each version of a system. Table 4 lists all methods we have taken into account to compute the *NMethodCall* metric. For instance, if an application places five method calls to method `startActivity`, the value for *NMethodCall* is 5. *NTry* counts the number of `try` blocks defined in the scope of classes that extend `Activity`, `Service`, `AsyncTask` and `Broadcast receiver` for each version of a system. Finally, *DensityTry* is defined as *NTry* divided by *NClasses*.

These metrics were computed using the eFlowMining tool (GARCIA; CACHO, 2011). Table 4 also presents the list of exceptions that these methods may throw, as informed by the Android documentation. The last two exceptions for the `Activity` type in this table are thrown by `startActivity` implementations of `android.support.v4` package used by apps. We do not explicitly count method calls related to intents because they are orthogonal to the aforementioned ones, e.g., one can use intents to send a broadcast, to start a service, etc.

3.1.2 Results and Analysis

This section reports our empirical findings and statistical tests of the hypotheses presented in Section 3.1.1.1. For the statistical tests performed along this section, we have employed the statistical package SPSS. We assume the commonly used confidence level of 95% (that is, p-value threshold = 0.05). The Spearman's rank correlation test is used to identify highly-correlated metrics. We use this test because in our analysis the metrics are nonparametric. For evaluating the results of the correlation tests, we adopted the Hopkins criteria to judge the goodness of a correlation coefficient (HOPKINS, 2013): < 0.1 means trivial, 0.1-0.3 means minor, 0.3-0.5 means moderate, 0.5-0.7 means large, 0.7-0.9 means very large, and 0.9-1 means almost perfect. The results of the correlation tests are presented in the next sections.

3.1.2.1 Android Abstractions and Changes to Exception Handling Code

Android does not propagate exception among `Activities`, `Asynctasks`, `Services`, and `Broadcast receivers`. All these abstractions need to handle all their exceptions otherwise the Android platform will terminate the whole application. In this context, this section investigates how exceptions are handled in the *caller* and *callee*, e.g, the caller in Figure 5 is represented by method `onClick` at the `MainScreen` Activity whereas the callee is represented by the class `FileScreen`.

Most of the Method Calls Are Not Protected. We first look at the *caller* side by analyzing the number of method calls to Android abstractions that are protected by `try-catch` blocks. Table 5 lists, for each Android abstraction, the number (*NMethod-*

Table 5: Descriptive statistics of calls inside and outside of try/catch blocks for analyzed categories.

			Application Categories					
			Social Network	Communication	Miscellaneous	News Reader /Media	Total	Percentage of method calls
Type of Android Call	**Activity**	Inside Try	183/ 12.42%	206/ 13.51%	310/ 17.50%	123/ 10.54%	822/ 13.85%	5934/ 64.22%
		Outside Try	1290/ 87.57%	1318/ 86.48%	1461/ 82.49%	1043/ 89.45%	5112/ 86.14%	
	Service	Inside Try	6/ 4.91%	77/ 12.60%	128/ 33.24%	18/ 6.97%	229/ 16.64%	1376/ 14.89%
		Outside Try	116/ 95.08%	534/ 87.39%	257/ 66.75%	240/ 93.02%	1147/ 83.35%	
	AsyncTask	Inside Try	67/ 23.18%	12/ 3.44%	0/ 0.0%	12 8.82%	91/ 8.75%	1040/ 11.25%
		Outside Try	222/ 76.81%	336/ 96.55%	267/ 100%	124/ 91.17%	949/ 91.25%	
	BroadCast	Inside Try	18/ 4.29%	22/ 13.66%	7/ 5.03%	19/ 11.11%	66/ 7.41%	890/ 9.63%
		Outside Try	401/ 95.70%	139/ 86.33%	132/ 94.96%	152/ 88.88%	824/ 92.58%	

Call) and percentage of method calls to the methods listed in Table 4 that are protected (*Inside Try*) or not protected (*Outside Try*) by `try-catch` blocks. An analysis of the last column in Table 5 reveals that method calls to activities are the most frequent ones for all categories and represent 64.22% of the cases. This is to be expected, since activities implement UI operations. Moreover, we found that 86.14% of the method calls to `startActivity` are not protected by `try-catch` blocks. This percentage is consistently spread throughout all applications categories. The second most frequent abstraction is `Service` with 14.89%. `Service` is also the abstraction with the highest percentage (16.64%) of protected method calls. In contrast, `AsyncTask` (with 8.75%) and `BroadCast` (with 7.41%) are the abstractions with the lowest percentages of method calls protected by `try-catch` blocks.

Reduced number of `try` blocks. One possible reason for not protecting Android method call invocation is that developers may know exceptions thrown by Android abstractions are not propagated back to the point where the method was invoked (*caller* side). Based on that, we now investigate whether developers are protecting the *callee* side from exception occurrences. Table 6 lists the average value of *DensityTry* per Android abstraction. For instance, the first row shows that there is less than one (0.59) `try` block per class that extends `Activity` in the Social Network category. When we consider all application categories, there are 2.641 classes that extend `Activity` against 1.711 `try`

Table 6: Statistics of density by abstractions

					Total			
	Social Network	Communication	Miscellaneous	News Reader /Media	Mean	Num of Classes	Num of Try	Percentage of Classes with Try
Activity	0.59	0.87	0.62	0.46	0.65	2641	1711	30.02%
AsyncTask	0.81	0.85	0.67	0.46	0.77	1862	1432	58.43%
BroadCast	0.25	0.18	0.29	0.11	0.22	832	183	14.66%
Service	0.98	2.32	0.86	1.87	1.54	868	1333	39.97%
Total	0.69	0.96	0.62	0.76	0.75	6203	4659	37.88%

blocks. This produces a total average of 0.65 `try` block per `Activity`. It is a very low density when considering that Android abstractions, such as Activities, have many entry points and multiple `try-catch` blocks would be expected to protect them from crashing the whole application.

The last column of Table 6 shows the average percentage of Android abstractions that have at least one `try` block. For instance, only 30.02% of the analyzed Activities have at least one `try` block. Overall, only 37.88% of the classes that extend Android abstractions have at least one `try` block. This means that 62.12% of the classes that extend Android abstractions have no exception handling code.

Insignificant variation in the number of `try` blocks per class. Figure 8 presents the value of the *DensityTry* measure, i.e., the number of `try` blocks per class, for each analyzed version of each subject applications. Each graph presents the data for the programs in each application category. There is little variation in the value of the metric in the versions of 13 out of the 16 subject applications. The exceptions are: *AndStatus* application had a slight decrease, *VLCAndroid* application presented a slight increase and *ConnectBot* had a moderate to significant increase in the *DensityTry* metric.

Figure 9 depicts an example for such stability. This figure shows the evolution in the exception handling code for the `startFromActivity` method, over four versions of the *AndStatus* application. From version 6.3 to 7.0, the developer added a `try` block with a specific handler for `android.content.pm.Pack-ageManager.NameNotFoundException` checked exception. This method did not change for version 7.1 and the `try` block was removed for version 8.0. It is possible to see that

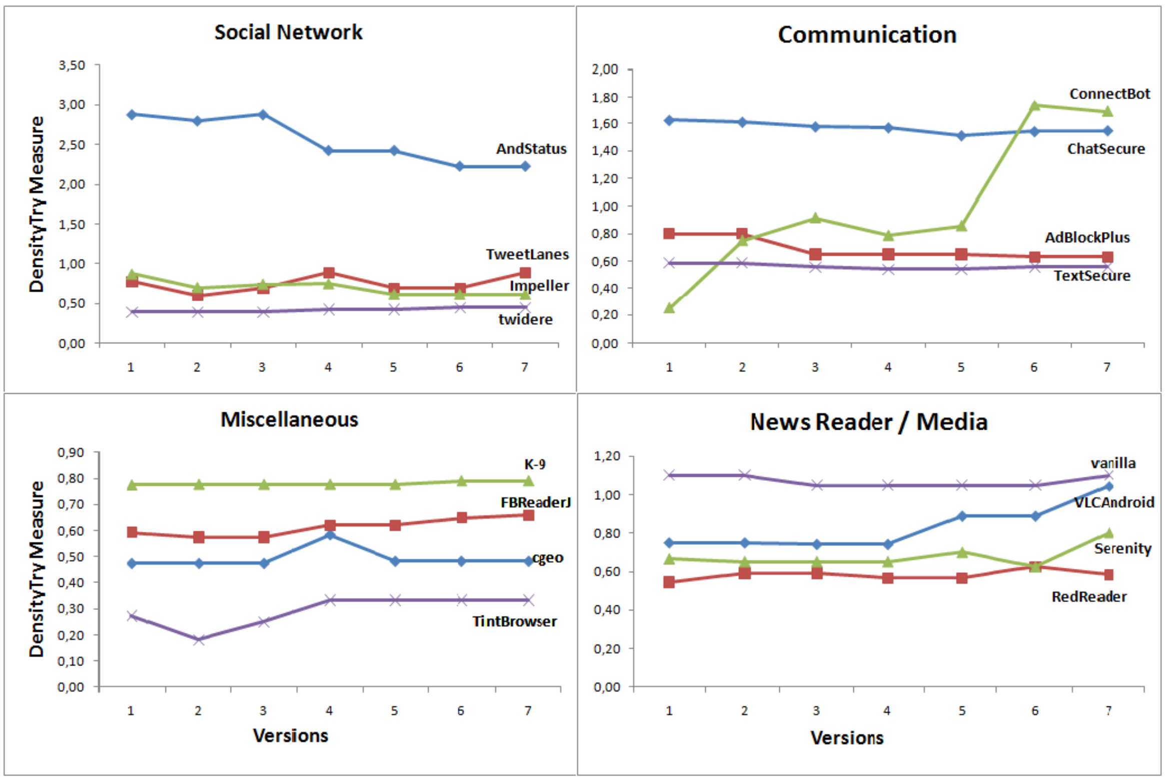
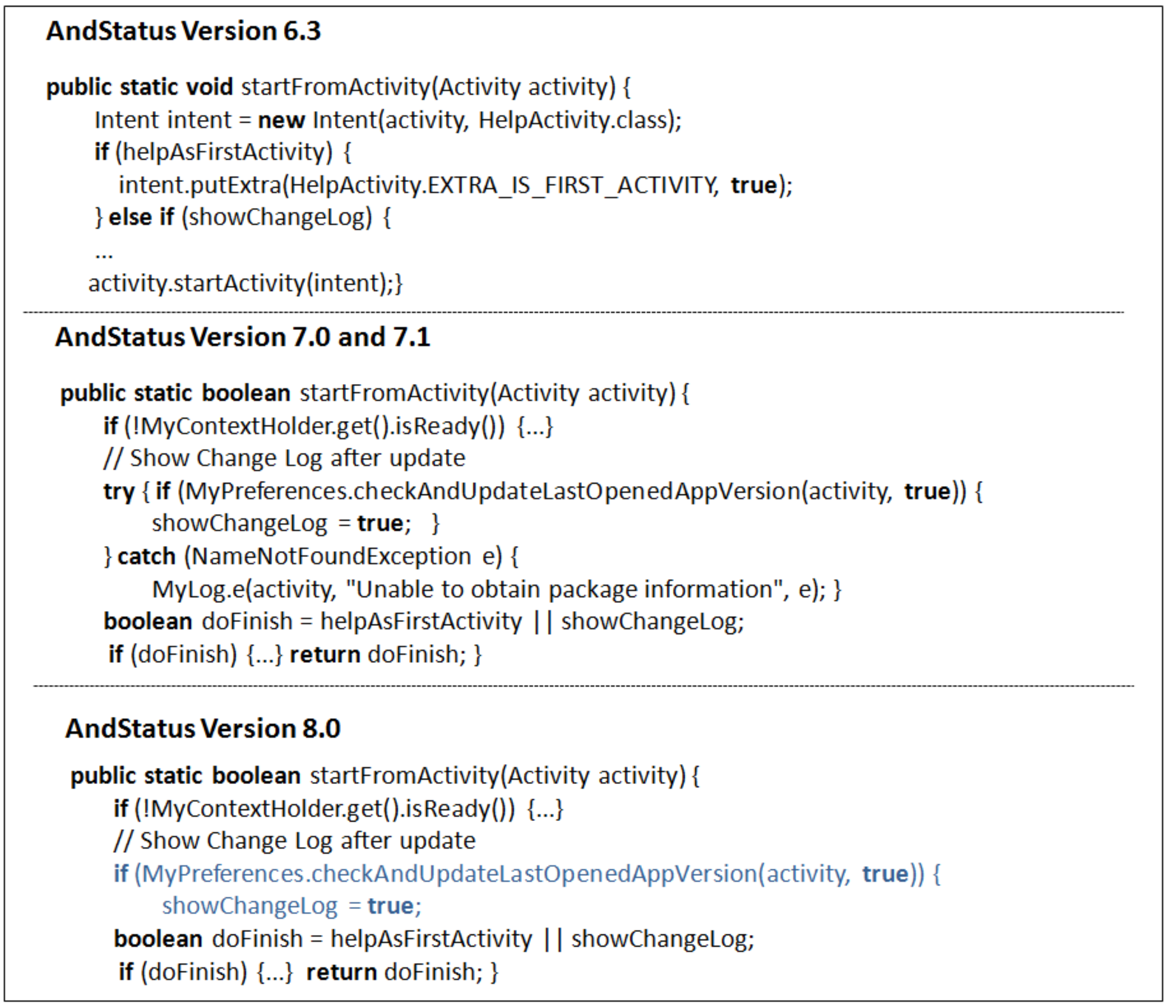


Figure 8: Values of the *DensityTry* measure throughout the versions.

the developer did not declare `NameNotFoundException` in the exception interface of the method nor treats it in any other version of the application.

In order to understand in greater depth the reasons for this apparent stability in the value of *DensityTry*, it was necessary to manually inspect all classes that extend Android abstractions (Activity, Services, AsyncTask and Broadcast) and changed (or were added) during the evolution of the application. In particular we inspected 220 change scenarios encompassing addition, modification and deletion of `try-catch` blocks. We observed and distinguished: (i) recurring co-changes between the normal code blocks and exception handling blocks, and (ii) independent changes made to either of them. Each change scenario was classified according to the action taken on the normal code (`try` block only) and exception handling code (`catch` block), following the classification presented in Section 3.1.1.3. Table 7 summarizes the frequency of change scenarios involving co-changes between the normal (columns) and exceptional (rows) code. The frequency of independent changes made to the normal code is captured in the row “No changes to catch blocks” whereas independent changes to the exceptional code are captured in the last column “No changes to try blocks”. The bottommost row, labeled “Percentage for Normal Code Changes”, shows the distribution of the frequency of normal change scenarios.

More modifications than additions of `try` blocks. An analysis of the last row



AndStatus Version 6.3

```
public static void startFromActivity(Activity activity) {
    Intent intent = new Intent(activity, HelpActivity.class);
    if (helpAsFirstActivity) {
      intent.putExtra(HelpActivity.EXTRA_IS_FIRST_ACTIVITY, true);
    } else if (showChangeLog) {
    ...
    activity.startActivity(intent);}
```

AndStatus Version 7.0 and 7.1

```
public static boolean startFromActivity(Activity activity) {
    if (!MyContextHolder.get().isReady()) {...}
    // Show Change Log after update
    try { if (MyPreferences.checkAndUpdateLastOpenedAppVersion(activity, true)) {
        showChangeLog = true;  }
    } catch (NameNotFoundException e) {
        MyLog.e(activity, "Unable to obtain package information", e); }
    boolean doFinish = helpAsFirstActivity || showChangeLog;
     if (doFinish) {...} return doFinish; }
```

AndStatus Version 8.0

```
public static boolean startFromActivity(Activity activity) {
    if (!MyContextHolder.get().isReady()) {...}
    // Show Change Log after update
    if (MyPreferences.checkAndUpdateLastOpenedAppVersion(activity, true)) {
        showChangeLog = true;
    boolean doFinish = helpAsFirstActivity || showChangeLog;
     if (doFinish) {...}  return doFinish; }
```

Figure 9: Example of Evolution in Handling an Exception in the AndStatus Application.

of Table 7 reveals that changes in the `try` blocks represent the most common change scenarios. The most common kinds of modifications involve changes to variables inside existing `try` blocks (fifth column), 19.64% of the cases, and modifications in the program control flow (sixth column), 15.47% of the cases. However, for those scenarios involving changes in `try` blocks, developers usually perform no change to handlers. These are cases where the `try` block was modified but not an accompanying `catch` block. For instance, no change to handlers occurs in: (i) 87.87% of the cases where variables were modified, and (ii) 80.76% of the cases here the control flow was modified. In fact, when we consider the overall distribution of exceptional change scenarios (last column), 37.3% of them did not involve changes to handlers in the Android apps (*No changes to catch blocks*). This is potentially dangerous because new methods and modifications to the program control flow may result in new exceptions being thrown that the associated `catch` blocks cannot capture. The actions performed by the `catch` blocks may help to explain this lack of change. In fact, a recent study of Nakshatri and colleagues (NAKSHATRI; HEGDE;

Table 7: Classification of Normal and Exceptional Change Scenarios related to Activity abstraction

Changes in the exceptional code	**Changes in the normal code (try blocks only)**									
	Try block added to existing method	New method added with try block	Try block removed	Method with try block removed	Variables modified	Control flow modified	Method call replaced	Intent call added	No changes to try blocks	Percentage for Exceptional Code Changes
Generic Handler added	21.42%	30.30%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	8.52%
Empty Generic Handler added	14.28%	3.03%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.97%
Generic Handler removed	0.0%	0.0%	47.82%	30.0%	0.0%	0.0%	0.0%	0.0%	0.0%	8.33%
Specialized Handler added	39.28%	54.54%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	16.66%	17.85%
Empty Specialized Handler added	14.28%	12.12%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	4.76%
Specialized Handler added to rethrow exception	10.71%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.78%
Specialized Handler removed	0.0%	0.0%	52.17%	70.0%	0.0%	0.0%	0.0%	0.0%	0.0%	11.30%
Reference change	0.0%	0.0%	0.0%	0.0%	0.0%	3.84%	0.0%	0.0%	0.0%	0.59%
Changing the Exception Handling Policy	0.0%	0.0%	0.0%	0.0%	12.12%	11.53%	12.5%	0.0%	16.66%	5.35%
Changing the catch block to use normal code	0.0%	0.0%	0.0%	0.0%	0.0%	3.84%	0.0%	0.0%	16.66%	1.19%
No changes to catch blocks	0.0%	0.0%	0.0%	0.0%	87.87%	80.76%	87.5%	100.0%	50.0%	37.30%
Percentage for Normal Code Changes	16.66%	19.64%	13.69%	5.95%	19.64%	15.47%	4.76%	0.59%	3.57%	

THANDRA, 2016) analysed 554,864 Java project in the GitHub repository and 50,692 Java project in the SourceForge repository. They observed that the most common action performed by `catch` blocks are: `printStackTrace`, log methods and empty blocks. Similar result was observed by Kery and colleagues (KERY; GOUES; MYERS, 2016) who analysed nearly 8,000,000 Java repositories and perceived that the most common action performed by `catch` blocks are: empty blocks (12.4%), print statement (10%), and log methods (10%). These results suggest that with small or no code the `catch` blocks will poorly evolve from one version to the other.

For instance, a common change scenario in the exceptional behavior occurs when `try` blocks are added. They are either added to existing methods (first column), in 16.66% of the cases, or together with the addition of new methods (second column), in 19.64% of the cases. However, for those scenarios involving new `try` blocks, developers usually attached poor exception handlers. For instance, generic handlers were introduced in: (i) 30.30% of the cases of new methods, and (ii) 21.4% of the cases of new `try` blocks for existing methods. The addition of `try` blocks to existing methods is also performed together with other forms of handlers: (i) addition of empty generic handlers (14.2% of the total), and (ii) addition of empty specialized handlers (14.28% of the total).

Testing hypotheses 1 and 2. In order to assess the first and second hypotheses, we analyze the data presented in Table 8. Its last column [4] shows the overall result of

[4]Correlations at the 95% level are marked with * in Table 8, while those at the 99% level are marked

Table 8: Spearman rank correlation between changes metrics, call metrics and calls related to abstractions.

		Application Categories				
		Social Network	Communication	Miscellaneous	News Reader / Media	Total
EHChurned LOC	NMethodCall	0.207	0.179	0.309	-0.154	0.133
	NClasses	0.432*	0.320	0.161	-0.036	0.253*
Uncaught Flow	NMethodCall	0.674**	0.922**	0.472*	0.782**	0.771**
	NClasses	0.762**	0.929**	-0.044	0.700**	0.783**
	DensityTry	-0.729**	-0.421*	-0.791**	-0.397	-0.507**

the correlations when considering all applications, regardless of program category. The first row shows that there is no significant correlation between the *EHChurnedLOC* and *NMethodCall* metrics. Thus, we are unable to refute Hypothesis 1. In other words, this result shows that when the *NMethodCall* is changed, exception handling code is usually not added or changed to handle possible exceptions propagated by the new method calls. The second row lists the correlations between the *EHChurnedLOC* and *NClasses* metrics (Section 3.1.1.3). This row presents only one statistically significant correlation. Considering that this correlation is only moderate (< 0.5) and no correlation was observed for the remaining application categories, we are unable to refute Hypothesis 2. This means that the data suggests a trend: when the number of *NClasses* changes, this change is not likely to affect the amount of exception handling code added or changed.

3.1.2.2 Android Abstractions and Robustness

The previous section reported that there is no significant correlation between exception handling code changes and the *NMethodCall* and *NClasses* metrics. This section discusses the relation between these two metrics and program robustness. We assessed if changes in the values of *NMethodCall* and *NClasses* are related to an increase (or decrease) in the number of uncaught exception flows. Table 8 presents the correlations between the metrics quantifying changes in the Android abstractions (NMethodCall and NClasses) and the metric for system robustness that we employ (Uncaught Flow).

Use of Android abstractions often decreases robustness. The analysis of Table 8 reveals that there is often a strong or very strong correlation between the use of Android

with **

abstractions (*NMethodCall* and *NClasses*) a the number of uncaught flows. The level of correlation varies according to: (i) the application category, and (ii) the type of change to exceptional code. The strongest correlations were observed in Android programs falling in the Communication category. In particular, for this category, there was an almost perfect correlation (> 0.9) between the use of Android abstractions (NMethodCall and NClasses) and uncaught exceptions. This type of correlation was also very strong (> 0.7) for the News Reader/Media category. As far as the Miscellaneous category is concerned, there is no correlation between changes in *NClasses* and the number of uncaught flows. On the other hand, there is a moderate correlation between *NMethodCall* and the number of uncaught flows.

Table 9: The Percentage of Uncaught Exceptions Generated by Change Scenarios

Scenarios	**Activity**	**AsyncTask**	**BroadCast**	**Service**	**Total**
No changes to catch blocks	38.9%	42.9%	-	-	32.3%
Specialized Handler added	22.2%	28.6%	-	33.3%	22.6%
Generic Handler added	5.6%	14.3%	66.7%	33.3%	16.1%
Specialized Handler added to rethrow exception	5.6%	-	-	-	3.2%
Empty Specialized Handler added	5.6%	-	-	-	3.2%
Generic Handler removed	-	14.3%	-	-	3.2%
Specialized Handler removed	16.7%	-	-	33.3%	12.9%
Changing the Exception Handling Policy	5.6%	-	33.3%	-	6.5%

Lack of changes to exception handling code leading to uncaught exceptions. An analysis per application category indicates that when method calls (NMethodCall) to and classes (NClasses) that extend Android abstractions are added without the addition of the corresponding exception handling code, there is usually an increase in the number of uncaught flows. Table 9 lists the change scenarios that most often generated uncaught exceptions. The *No changes to catch blocks* scenario was responsible for generating 32.3% of the uncaught exception flows (last column) and the most common one for `Activities` and `AsyncTask`.

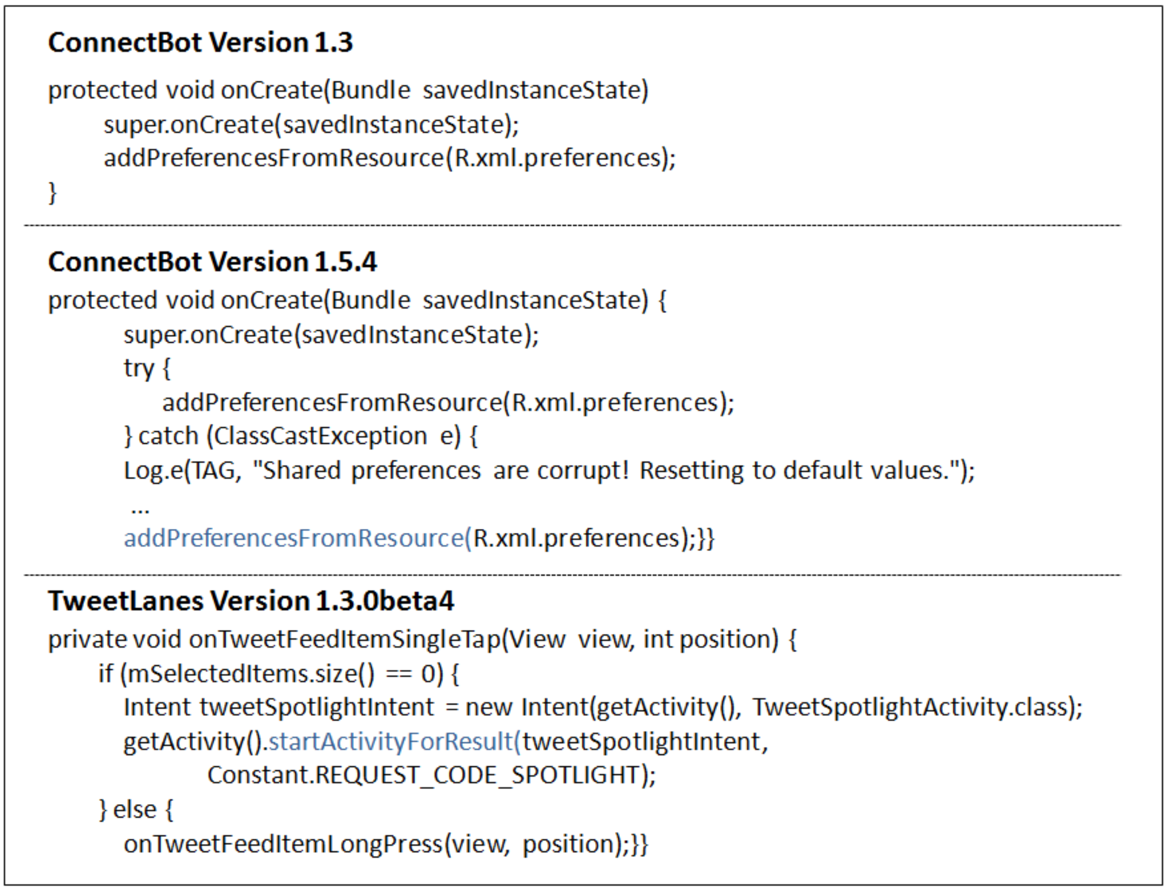
Method calls to Android abstractions usually propagate unchecked exceptions (see Table 4) and the Java compiler does not perform reliability checks for this type of exception. Hence, the developer cannot determine whether a `catch` block needs to be added or changed to handle exceptions without examining the documentation. For example, according to the documentation of class `Activity`[5], 12 public methods may throw `SecurityException`, 10 may throw `ActivityNotFoundException`, and 6 may throw `SendIntentException`. Among these exceptions, only the latter is checked. Any other exception thrown by these methods is necessarily also unchecked. Additionally, according to the documentation of the `AsyncTask` class[6], none of the methods listed in Table 4 throw checked exceptions, which means that any problem during the execution of an `AsyncTask` results in the throwing of an unchecked exception. Furthermore, the caller of an `AsyncTasks` must start its execution by invoking the methods `execute()` or `executeOnExecutor()`. Both may throw `IllegalStateException`, an unchecked exception. In the Android platform, unchecked exceptions thrown by entry point methods of Android abstractions (such as the `onCreate` method of the `Activity` class) are not received by the caller, generating the propagation of the exception to the platform. Unfortunately, in all the analyzed applications, we could not found any class implementing the `Thread.UncaughtExceptionHandler`. This interface defines handlers capable of handling exceptions in threads that are about to terminate due to an uncaught exceptions.

Type of changes that decrease robustness. Even when the number of handlers increases over time, program robustness is reduced if exceptions are not properly bound to handlers. For instance, there is moderate, significant correlation between *EHChurned-LOC* and *NClasses* for the Social Network category. However, there is also very strong, significant correlation between *Uncaught Flow* and *NClasses* for the same category. We observed that certain types of change unexpectedly led to uncaught exception flows. For instance, Table 9 shows that the addition of `catch` blocks seemed to increase the number of uncaught flows. In particular, this was the case when adding specific handlers. This case represented 22.6% of the uncaught exception flows (last column of Table 9). This result seems to be related to the low frequency of exceptions declared in the exceptional interface of methods implementing Android abstractions (COELHO et al., 2015). In fact, outdated exception interfaces affect program comprehension, since it is not possible to know the exceptions that a method may throw just by looking at its exception interface. However, Java programmers are used to relying on exception interfaces to obtain this information. As a consequence, when they implement code to invoke methods with outdated exception interfaces, if they only provide handlers for the exceptions in the interfaces, new uncaught exceptions may flow through the system.

[5]http://developer.android.com/reference/android/app/Activity.html

[6]http://developer.android.com/reference/android/os/AsyncTask.html

Testing hypotheses 3 and 4. The last column of Table 8 shows the overall result of the correlations when considering all applications, regardless of category. The results show that all correlations are very strong and significant. In other words, when using Android abstractions (*NMethodCall* and *NClasses*) such usage may be reflected in the robustness of the system. Specifically, when exceptional handlers are not properly employed to protect the utilization of Android abstractions, there is an increase in the number of uncaught flows. Therefore, we reject Hypotheses 3 and 4. Finally, we also observed a negative and moderate (-0.507) correlation between *DensityTry* and uncaught flows. This means that when developers increase the number of try blocks per class, the number of uncaught flow decreases.



ConnectBot Version 1.3

```
protected void onCreate(Bundle savedInstanceState)
      super.onCreate(savedInstanceState);
      addPreferencesFromResource(R.xml.preferences);
}
```

ConnectBot Version 1.5.4

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        try {
            addPreferencesFromResource(R.xml.preferences);
        } catch (ClassCastException e) {
        Log.e(TAG, "Shared preferences are corrupt! Resetting to default values.");
         ...
        addPreferencesFromResource(R.xml.preferences);}}
```

TweetLanes Version 1.3.0beta4

```
private void onTweetFeedItemSingleTap(View view, int position) {
      if (mSelectedItems.size() == 0) {
        Intent tweetSpotlightIntent = new Intent(getActivity(), TweetSpotlightActivity.class);
        getActivity().startActivityForResult(tweetSpotlightIntent,
                 Constant.REQUEST_CODE_SPOTLIGHT);
      } else {
        onTweetFeedItemLongPress(view, position);}}
```

Figure 10: Example of Changes that Affect the Number of Uncaught Flows.

Figure 10 shows two examples of changes that affect the number of uncaught flows in *ConnectBot* and *TweetLanes* applications in classes of type `Activity`. According to figure, we can see that there was an addition of a `try` block with a handler specific to the `ClassCastException` exception, in version 1.5.4 of the *ConnectBot* application. In this handler was added code for the exception handling, where the `addPreferencesFrom-Resource` method of the `android.preference.PreferenceActivity` class is called again in the catch block. There were, for this couple of versions of the application, an increase of 22 uncaught flows for the `java.lang.RuntimeException` exception. This exception is thrown explicitly by the method called within the code of the `addPreferencesFromResource` method and is not being handled in this example. In Figure 10 we also see an example call to the `startActivityForResult` method in the *TweetLanes* application. The `onTweetFeedItemSingleTap` method calls this method without adding a suitable handler. This method explicitly throws the `java.lang.IllegalArgumentException` exception, which is not being handled in this example. These examples support rejection of hypotheses 3 and 4 as they show that the use of Android abstractions may reflect the robustness of applications.

3.1.3 Threats to Validity

This section discusses threats to validity that can affect the results reported in this research work.

Internal Validity: Threats to internal validity are mainly concerned with unknown factors that may have had an influence on the experimental results (WOHLIN et al., 2000). To reduce this threat, we have selected a set of subject applications whose developers had no knowledge that this study was being performed.Additionally, our results can be influenced by the performance, in terms of precisions and recall, of the used tools. We tried to limit the number of false positive through a manual validation. The use of this validation mitigates threats to internal validity, but does not completely remove them as a certain amount of imprecision cannot be avoided by the tools that generally deal with undecidable problems. Likewise, the change metrics collected manually were validated by one researcher who was not aware of the experimental goal.

Construct Validity: Threats to construct validity concern the relationship between the concepts and theories behind the experiment and what is measured and affected (WOHLIN et al., 2000). To reduce these threats, we use metrics that are widely employed to quantify the extent of changes over the software evolution (YAU; COLLOFELLO, 1985; NAGAPPAN; BALL, 2005; GIGER; PINZGER; GALL, 2011) and to measure the software robustness (ROBILLARD; MURPHY, 2003; FU; RYDER, 2007; COELHO et al., 2008; MAJI et al., 2012).

External Validity: External validity issues may arise from the fact that all the data is collected from 16 software systems that might not be representative of the industrial practice. However, the heterogeneity of these systems helps to reduce this risk. They are implemented in Java, which is a representative language of the state of object-oriented programming practice. Furthermore, the characteristics of the selected systems, when contrasted with the state of practice, represent a first step towards the generalization of the achieved results.

It is fair to argue that a number of recent studies have targeted hundreds of systems (COELHO et al., 2015; KECHAGIA; SPINELLIS, 2014; LINARES-VáSQUEZ et al., 2013; OKUR; DIG, 2012; PINTO et al., 2015), which is in stark contrast to the 16 that we have analyzed. However, these studies are designed so that they can be performed in a completely automated manner. This is inherent to the data they aim to collect, e.g., syntactic information about usage of specific constructs (PINTO et al., 2015) or stack traces (COELHO et al., 2015). However, it is impractical to generalize the collection of information about exception handling change scenarios, e.g., Changing the `catch` block to use normal code (Table 2). Moreover, we have performed exception flow analysis on these systems. This kind of analysis requires all the dependencies of a system to be resolved and is expensive to conduct. It is also important to mention that we have analyzed 112 versions of these systems. Each one of these versions had to be analyzed both in isolation and in comparison to previous versions. These factors hint at the depth of the analysis that was performed in this study but also explain the limited number of systems that we are able to analyze. Finally, other studies (CABRAL; MARQUES, 2007; CACHO et al., 2014; ZHANG; ELBAUM, 2012) that analyzed exception usage and also employed a mixture of manual and automated program analysis also targeted a relatively small number of systems and some of them (CABRAL; MARQUES, 2007; ZHANG; ELBAUM, 2012) targeted only a single version.

Reliability Validity: This threat concerns the possibility of replicating this study. The source code of the subject applications is publicly available. The way our data was collected is described in detail in Section 3.1.1.3. Moreover, the eFlowMining tool is available to obtain the same data. Hence, all the details about this study are available elsewhere to enable other researchers to control it.

3.2 Energy Consumption of EH Strategies

Energy consumption is an important non-functional requirement for mobile applications that execute in devices with a limited battery capacity. Some studies aim to investigate energy consumption of specific device components like Screen, GPS, Network

(LI et al., 2014), and others investigate that at the code level (LI et al., 2013), approach level (OLIVEIRA; OLIVEIRA; CASTOR, 2017) or API level (LINARES-VÁSQUEZ et al., 2014). The work of Chowdhury, Shaiful, et al. (CHOWDHURY et al., 2018) analyzes energy consumption for the common strategy of Log information used widely by developers and Zhang, Jack, Ayemi Musa, and Wei Le (ZHANG; MUSA; LE, 2013) evaluate bugs that cause reduced battery life. To the best of our knowledge, no study has set out to investigate whether the use of exception handling strategies has an impact on energy consumption of Android applications. In this context, the following study aims to investigate the power consumption of the applications using specific exception handling strategies, to see if there is an impact on the power consumption.

3.2.1 Study Setting

3.2.1.1 Goal Statement and Study Hypothesis

The purpose of this study was to identify if there is a difference in the energy consumption of the applications when executed with and without exception handling. In the experiments, the focus is on running components by performing specific exception handling activities or by running normally. The definition of our hypotheses uses the following metrics collected from the power consumption experiments: PercentageUsage, is the percentage of battery used by the application; BatteryLevel, is the battery level in percentage at the end of the experiment and Power, is the amount of energy consumed by the application. In this study this metric is measured in Watts (W). The main hypothesis is the *null* hypothesis that states there is no difference of energy consumption for applications that do not use the EH strategies and applications that use the EH strategies. There are three null hypotheses, one for each metric of interest.

- **Null hypothesis** ($H0_1$): The PercentageUsage distribution is the same in the Strategy categories.
- **Null hypothesis** ($H0_2$): The BatteryLevel distribution is the same in the Strategy categories.
- **Null hypothesis** ($H0_3$): The distribution of Power is the same in the Strategy categories.

3.2.1.2 Target Applications

We select a set of ten Android applications with different characteristics, developed to different domains. Fala Natal is an application for citizen engagement. K9mail is an mail application where the user can manage various types of email accounts. Bitcoin Wallet is a bitcoin application where the user can send and receive bitcoins via NFC, QR codes or using bitcoin addresses, make payments via Bluetooth. MediaPlayer is an application that offers the functionalities of a music player for the user. NotesList and Contacts are respectively, applications to notes management and contacts management on the smartphone. Cafe is an application that the user can find coffee places nearby, using the map. BluetoothChat is an application that allows the user to control the Bluetooth of the smartphone and AndroidSecurity is an application that the user can use to do secure authentication. Finally, VisitNatal, that is a mobile tourist guide application implemented in Java for Android Platform that aims to enhance tourists' travel experience. Visit Natal is the official tourist guide application for the city of Natal in Brazil.

3.2.2 Experiments Execution

After selecting the set of applications, three exception handling strategies were defined to be analyzed in the experiments: Error log, error display using Android Toast, and save the error message in a file. These strategies were manually inserted in the code of the applications for lifecycle methods like `onCreate` and event-handlers methods like `onClick`. For each method, a try/catch block was added throwing a runtime exception and handling this exception with the selected strategy. We inserted the strategies in the code of classes that represents the main functionalities of the application. We use Monkey tool to generate random events in the user interface. We created a seed value for the pseudo-random number generator (700). The seed value generated the same sequence of events for all applications.

The experiments were performed as follows: Each application was executed four times, once without using any exception handling strategy and the other three times using the strategies; the initial smartphone battery level must be above 60%; the interaction events with the user interface were generated automatically by the Monkey tool; in each experiment the Monkey was executed 380 times, generating 1000 random events for each execution; The experiments were executed on the Samsung Galaxy S7 edge device; energy consumption data obtained through `batterystats` tool from Android and visualized with Battery Historian tool. In total, 10 experiments were performed without exception

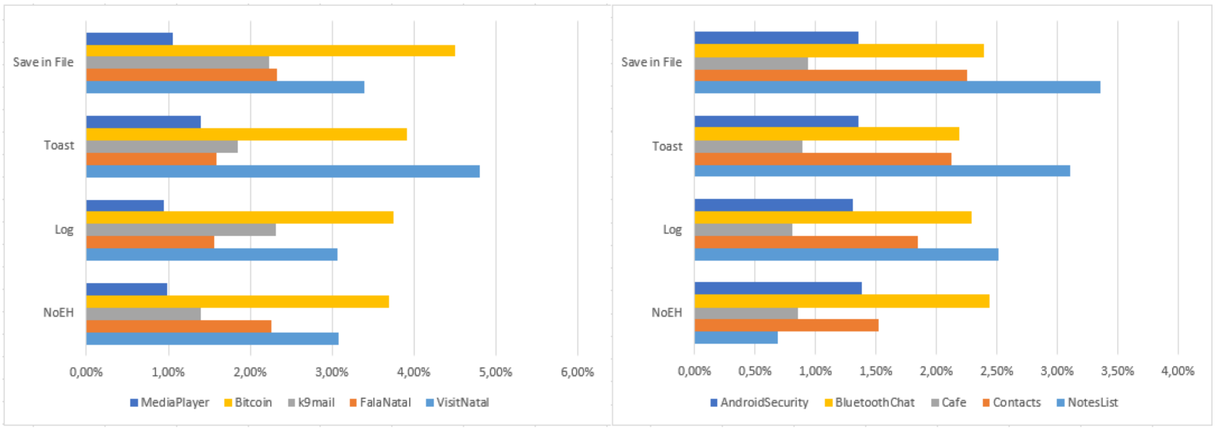


Figure 11: Percentage of battery usage by applications

handling and 30 experiments were performed with the use of exception handling strategies.

3.2.3 Study Results

From the experiments, data were collected such as: battery level at the beginning and end of the experiments, percentage of battery use by the application, initial and final levels of the battery in mAh, voltage and initial and final time of the experiments. From these data, energy consumption in Watts can be calculated. This section shows the results obtained from the execution of the experiments.

Figure 11 shows data on the percentage of battery usage by applications and by type of strategy. We can see that in general when the application is executing exception handling to save the error data in a file, the percentage of battery usage by the application is higher, indicating that this strategy can be more costly in terms of energy consumption in relation to the others. Analysis of graphics indicates that the strategy of showing a message with the error using Android's Toast feature is more costly than using just the Log in most cases, except for the k9mail and BluetoothChat applications. These graphics demonstrates that in most cases the addition of exception handling in the application resulted in a greater use of the battery during the experiments. Only in some cases, FalaNatal and BluetoothChat applications for example, the use of Log or Toast did not result in an increase in this percentage.

Figure 12 shows the graphics with energy values (power, W) in Watts consumed by the device during experiments by applications and by type of strategy. Looking at the graph it can be seen that in most applications the experiments with exception handling resulted in an increase in energy consumption, with some exceptions like the K9mail and NotesList applications and some cases in which the use of Log or Toast did not result

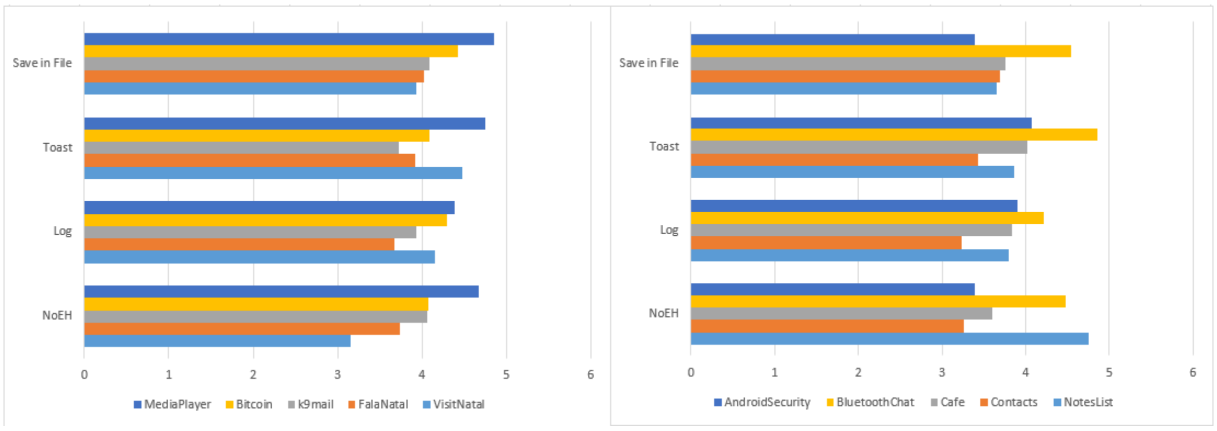


Figure 12: Power consumption by applications

in an increased energy consumption. It can be noted that in half of the applications the experiments that used the strategy of saving the error in a file consumed more energy than using the strategy with Toast and in eight applications, the vast majority, using the exception handling strategy with Toast consumed more energy than using just the Log. Saving the error to a file also consumed more energy than using the log in most applications.

In order to assess the hypotheses whether the results for percentage of battery usage, battery level and power have significant difference we need to check first whether the data is parametric or not, this is important to decide the statistical method that will be used. For this purpose we need to check if the data follows the normal distribution and to verify the homogeneity of variance. To perform the statistical analysis of the collected metrics we use the statistical package SPSS. We assume the commonly used confidence level of 95% (that is, p-value threshold = 0.05).

We use two grouping variables for the data: NoEH and EH, the first indicates the values collected from the experiments where the applications do not use EH strategies, the second one, represents the values when the applications are using EH strategies. Table 10 shows the two normality tests employed by SPSS, Kolmogorov-Smirnov and Shapiro-Wilk for the data. The results show that there is no statistically significant difference for the metrics (column Sig. of Shapiro-Wilk), so we can assume that the data follows the normal distribution.

Verifying that the data follows a normal distribution is not enough to verify whether they are non-parametric or not. It is necessary to perform a second test to verify the homogeneity of variance, if there is a statistically significant difference between the data in this second test, the data will be non-parametric, otherwise, they will be parametric

Table 10: Normality Tests of the study

		Kolmogorov-Smirnov			Shapiro-Wilk		
	Strategy	**Statistic**	**gl**	**Sig.**	**Statistic**	**gl**	**Sig.**
PercentageUsage	**NoEH**	0,221	10	0,184	0,917	10	0,331
	EH	0,184	10	0,200*	0,936	10	0,507
BatteryLevel	**NoEH**	0,184	10	0,200*	0,948	10	0,639
	EH	0,179	10	0,200*	0,962	10	0,807
Power	**NoEH**	0,131	10	0,200*	0,932	10	0,469
	EH	0,221	10	0,181	0,942	10	0,575

Table 11: Homogeneity of Variance Test of the study

		Levene	gl1	gl2	Sig.
PercentageUsage	**Based on average**	7,069	1	18	0,016
	Based on median	5,692	1	18	0,028
	Based on median and adjusted gl	5,692	1	11,163	0,036
	Based on average trimmed	6,825	1	18	0,018
BatteryLevel	**Based on average**	11,539	1	18	0,003
	Based on median	10,869	1	18	0,004
	Based on median and adjusted gl	10,869	1	12,833	0,006
	Based on average trimmed	11,562	1	18	0,003
Power	**Based on average**	4,548	1	18	0,047
	Based on median	1,755	1	18	0,202
	Based on median and adjusted gl	1,755	1	10,985	0,212
	Based on average trimmed	4,476	1	18	0,049

data. Table 11 shows the results of the homogeneity of variance test for the metrics. The results show that there is a statistically significant difference between them (column Sig.), which means that the data is non-parametric.

Based on this result we select the non-parametric Kruskal-Wallis test to assess the three null hypotheses of the study. The Figure 12 shows the results of the Kruskal-Wallis test for the three metrics of the study. The results show that there is a statistically significant difference in energy consumption in terms of percentage usage of energy by the application, battery level, and power. Based on these results *we reject the three null hypothesis ($H0_1$, $H0_2$ and $H0_3$).*

The EH strategies used in this study are commonly used by developers when they want to handle exceptions and are not so complex to implement. It is important to mention that if a developer uses one or more of these strategies in a few places in the application

Table 12: Kruskal-Wallis Test of the study

	PercentageUsage	BatteryLevel	Power
H of Kruskal-Wallis	12,623	14,318	14,286
df	1	1	1
Significance Sig.	0,0001	0,0001	0,0001

that in the running time are not activated or rarely activated, this will not contribute to increase the energy consumption of the application. However, these results indicate that it is important to evaluate the impact on energy consumption at this level, especially in cases where the EH strategy is more complex and requires more processing or in cases where an error can occur repeatedly in the application. So, during the application development if a developer needs to make a trade-off between reliability and energy efficiency maybe it is interesting considering EH strategies used in the application, not only the behaviour of the components.

3.3 Conclusion

These two studies showed that in fact, exception handling has an impact on robustness and energy consumption of Android applications. Energy consumption has been a concern for developers of Android applications, the platform encourages developers to develop applications that optimize energy consumption, providing tools for inspection of energy use, such as Energy Profiler, and energy consumption analysis such as Battery Historian, as well as optimization tips for implementing applications. Developing robust applications is also important to provide applications that are reliable for users. Some studies has shown that Android developers do not use the platform's EHM properly and that the crash of applications is the biggest cause of complaints among users.

The results presented in this chapter showed the need to propose an approach that takes into account the relationship between the use of exception handling with other non-functional requirements in Android applications. This solution needs to guide the developer to think about these factors while developing, not just the features that need to be developed.

4 Proposed Methodology to Engineering Efficient Exception Handling for Android Applications

The previous chapter described two studies that showed how exception handling may impact on the robustness and energy consumption of Android applications. Two problems inherent from the EHM used by the Android platform were mentioned in the Section 1.1 of the introduction. Both are related to the standard strategy of the platform to deal with unchecked exceptions that are not handled in the application and the lack of a holistic view for the exception handling. Many of these exceptions can be thrown by Android abstraction methods, which can lead to an application crashing if these exceptions are unchecked and have not been properly handled. In addition, developers are not properly handling erroneous situations during development, section 3.1 points out that `UncaughtExceptionHandler`, a strategy encouraged by the Android platform, was not used in any of the applications. Energy consumption is a concern for developers of Android applications and users, and it is an important non-functional requirement to be considered when developing these applications. So the exception handling strategies that will be used in the application as well as the functioning of certain components must take into account the impact on energy consumption. This chapter aims to present a proposed methodology for efficient engineering exception handling for Android applications, which includes tools and a new exception handling mechanism to support the utilization of the methodology.

4.1 Proposed Methodology

This section describes a systematic methodology for designing efficient Android applications through an interactive process of exploring the possible design spaces and finding the efficient fault tolerance solutions. Its main contributions are as follow:

- a comprehensive methodology that supports trading off power consumption, reliability and resource usage during exception handling design,
- a selection of tools and guidelines that support the methodology, including the automated model generation,

Android applications are getting more complex than ever been before. They are everywhere, on everyone's mind, and in many different forms: social media, healthcare, security, government, and so on. Android developers have to balance diverse requirements such as connectivity, location, robustness, user-friendly interface and energy efficiency; this has given developer teams the hard task of choosing, without having a comprehensive methodology to handle many trade-offs, such as the one between application robustness and energy efficiency. To make such a choice, developers ask themselves the following questions: When and where should I employ exception handling for my Android application? What is the impact on energy efficiency? What is the impact on application performance? All these questions are running around in the minds of most Android developers and this work aims to help identify the places where exception handling should be used and its impact on energy efficiency and performance.

4.1.1 Proposed Workflow

An overview of the proposed workflow is shown in Figure 13, the blue boxes represent the activities that are performed automatically using the tools that support the methodology. The gray boxes represent the activities that will require manual work by the person responsible for the application, the green boxes represent sub-activities, and finally the white boxes represent the artifacts generated and used in each activity. Archon (RAFIEV et al., 2014) is a modelling framework that has an implementation that uses resource-driven graph representation to represent the application components and resources. For systems whose design domains are organized into multiple layers or levels, Archon facilitates the analysis and potentially its design and synthesis. Using Archon simulations developers or system designers can reasoning about robustness, energy consumption, and performance. The following subsections explain in more detail the workflow activities using the Visit Natal application as an example.

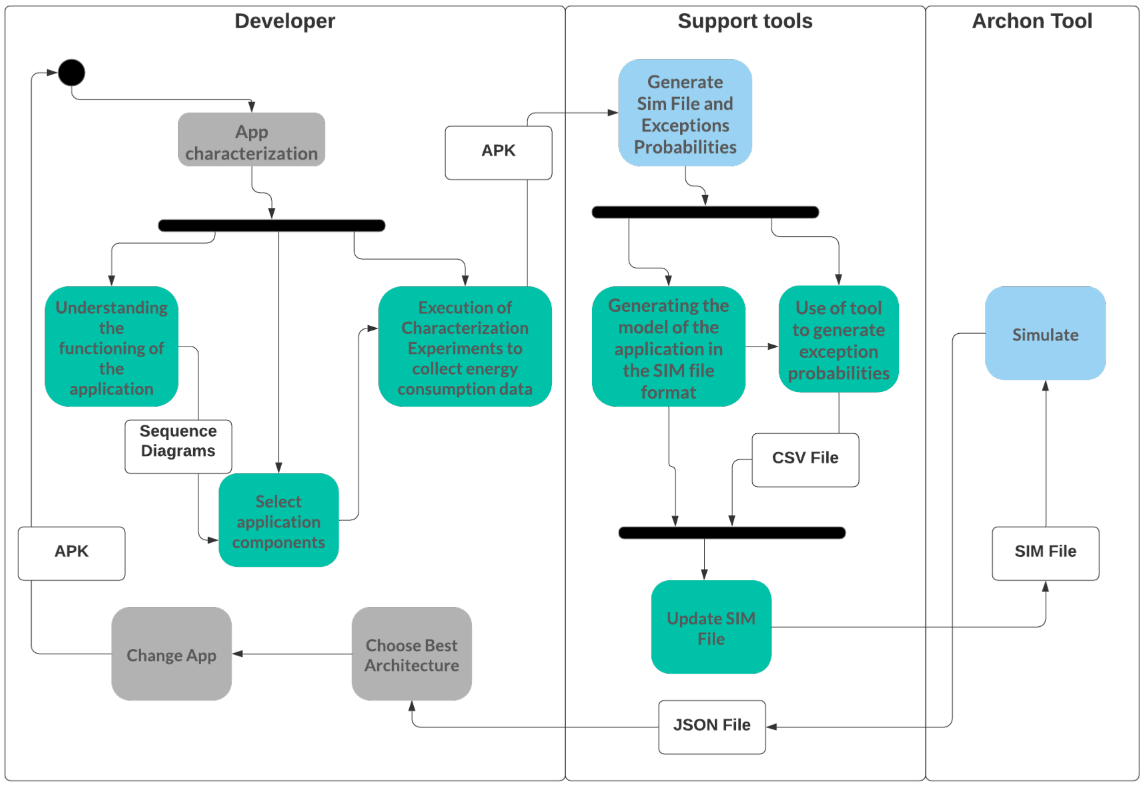

Developer
Support tools
Archon Tool
App characterization
APK
Generate Sim File and Exceptions Probabilities
Understanding the functioning of the application
Execution of Characterization Experiments to collect energy consumption data
Generating the model of the application in the SIM file format
Use of tool to generate exception probabilities
Simulate
Sequence Diagrams
CSV File
Select application components
APK
SIM File
Update SIM File
Change App
Choose Best Architecture
JSON File

Figure 13: Proposed Workflow

4.1.1.1 App characterization

The *App Characterization* is the first activity of the workflow and is executed by a developer or software engineer responsible for the application. This activity splits into three other subactivities: Understanding the functioning of the application, Select application components, and Execution of Characterization Experiments to collect energy consumption data.

The first subactivity consists in understanding the functioning of the application in terms of components and their interactions and main functionalities. This can be represented by sequence diagrams of the main functionalities. Figure 14 shows an example of sequence diagram generated for Visit Natal application, showing the interactions for the LocationService component. In the Visit Natal application scenario, background services such as LocationService can consume a lot of battery, so this is an scenario that can be analyzed using the methodology workflow.

In the Figure 14 we can see the components of the application that interacts of LocationService. The component MainFragment initialize the LocationService through a Intent call. This service runs continuously in background executing POST requests to

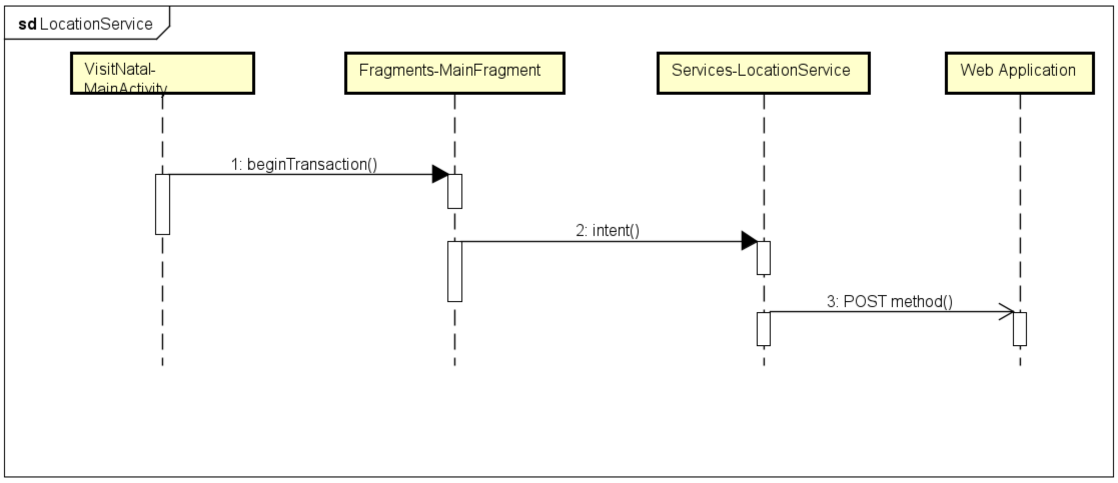


Figure 14: Sequence diagram for LocationService component of Visit Natal application

send the current location for an external component, the web application server. Knowing the functioning of the application, components can be selected to perform the experiments of energy consumption (*Select application components* subactivity). It is interesting select components that may affect the energy consumption of the application. Components that make use of GPS, camera, network, location, for example, may be candidates for the execution of energy consumption experiments.

Thr *Execution of Characterization Experiments to collect energy consumption data* subactivity consists in execution of experiments with the components for idle mode of the smartphone, change the application code to allow the component to run continuously, installation of the application and execution for a period of time to collect the data. All the data collected from these experiments are used in the activity, *Generate Sim File and Exceptions Probabilities.*

4.1.1.2 Generate Sim File and Exceptions Probabilities

The application bytecode is considered as input unit, guiding the development from the simulation to the implementation of the application. This allows to easily integrate our approach to any real-world development lifecycle. Based on the bytecode file (APK file) and the *app characterization* (Section 4.1.4), the *Generate Sim File and Exceptions Probabilities* activity is responsible for generating the model of the application in the SIM file format and *exceptions probabilities* (Sections 4.1.2 and 4.1.3). An example of Sim file generated for Visit Natal application, baseline version, can be seen in the Appendix B. An example of the SIM file format declaration for an application component can be seen as follows:

```
#assign MainActivity ".FaultyTask"
```

```
#setup MainActivity "type:userCmd; preDelay:2; postDelay:0;
power:14.63; pEx1:0.48; pEx3:0.02; pEx2:0.07"
```

The first declaration is used to assign an application component declaring it name and the type, in this example, the component `MainActivity` is a `FaultyTask` that has a probability of throwing exceptions. The second declaration is the setup of the application component, where its properties are defined.

The *Generate Sim File and Exceptions Probabilities* activity splits into two other subactivities: Generating the model of the application in the SIM file format and Use of tool to generate exception probabilities. These subactivities are executed by the support tools of the methodology and generate two artifacts: A CSV file with exception probabilities data and a SIM file that is the model of the application, with the components and interactions, used by Archon tool simulations. The data collected from characterization experiments related to power, battery and delay of components should be updated in the SIM file (*Update SIM File* subactivity) and also, the automatically generated exception probability data.

4.1.1.3 Simulate

After generating the SIM File for a specific version of the application, this model is simulated in the ArchOn tool for a fixed period of virtual time (model time) (Section 4.1.5). In the workflow this process is represented by *Simulate* activity. The simulation outputs a JSON file that includes average response time (ms), amount of energy (Joules) consumed, number of exceptions thrown, among other relevant information. The Appendix C shows an example of the JSON file generated by Archon tool.

4.1.1.4 Choose Best Architecture

These data generated by Archon tool are used in the next activity, *Choose Best Architecture*, to make design choice on which error handling strategies should be used in order to provide an improved balance between performance, energy, and reliability. To make design choices developers can run simulations that represent different behaviors of the application components according to the non-functional requirement (performance, energy, and reliability) that is a priority in a given scenario, comparing the results with the baseline simulation, for example, in the baseline simulation a component can execute without using any exception handling strategy, and in another simulation some strategy can be used. These simulations can represent different operation modes for the application.

4.1.1.5 Change App

Finally, application developers should code back the required changes to the application, build a new APK file, from which a new iteration of the workflow can be executed, this is represented by the activity *Change App* in the workflow. The developer uses DroidEH to implement the different behaviors on the components related to exception handling strategies and general component behaviour using the controller of DroidEH. App characterization is not required for the second and following iterations as the data from the first iteration should still be relevant.

4.1.2 Modelling

4.1.2.1 Resource Driven Modelling

The central subject of our method is the study of a computational platform comprising a number of diverse resources and the way resources may be handled in order to realize a computation. A resource is in this case an indivisible element required by the system in order to change its state, and it is defined by its function and availability in relation to this transition. With the word “resources" we make the point that we do not exclude computation, communication, or other facilities, e.g. energy and time. The work in (RAFIEV et al., 2014) proposes to represent a system with a relation graph, consisting of a set of nodes and a set of edges, where each node represents a single resource, and each edge represents a dependency between two resources.

Resource nodes also have a transient states that depend on the state of adjacent nodes. In the context of this work, the state of a resource represents resource activation, which can be used to track system energy and performance.

The ArchOn [1] tool is used to perform model simulation and analysis. Its features include a support for analyzing extra-functional properties at arbitrary abstraction levels. In this work, we implement and simulate abstract application models at the level of task graphs.

[1]https://github.com/ashurrafiev/ArchOn

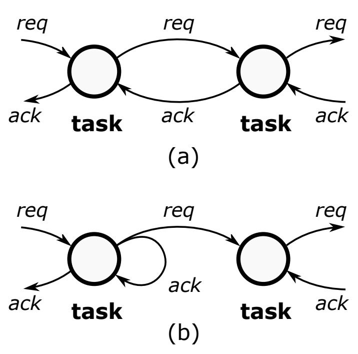


Figure 15: Using request-acknowledge protocol to model task interactions: (a) synchronous, (b) asynchronous.

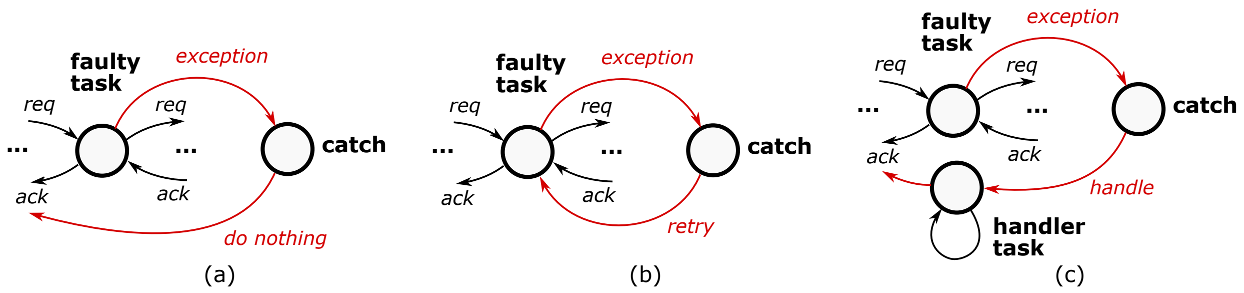


Figure 16: Modelling different exception handling scenarios: (a) do nothing, (b) retry task, (c) proceed to a recovery or an alternative task.

4.1.2.2 Modelling Patterns

The core resource element of the proposed application model is a **task**. Interactions between tasks are modelled using request-acknowledge protocol as shown in Figure 15. When a task is activated, it toggles a *req* to the next task after a delay. This delay represents the task performance and can be found by profiling the task experimentally as shown in Section 4.1.4. In synchronous interaction, the *ack* is toggled only after the *ack* from the next task is received; in asynchronous, *ack* is triggered instantly by the outgoing *req*. Synchronous interactions can provide an estimate for response time. Asynchronous interactions do not affect response time, but still are accounted for when calculating system energy consumption.

Certain tasks are not dependent on other tasks and require an external trigger. The proposed model provides two components that perform task invocation: **timer** and **user**. The timer resource calls a specific task at regular intervals, which represents a background service. The user (environment) resource attempts to mimic user behavior by doing stochastic task invocation with normally distributed random delays, which models GUI interactions. Additionally, the user model can emulate an application life cycle (start and stop); time intervals between the subsequent application launches are exponentially distributed.

In order to model fault-tolerance, we introduce additional model elements. The **faulty task** works like a regular task but has a probability to go to the *exception* branch instead of issuing the next request. This is the probability of a component going from normal state of operation to abnormal, and it is not affected by error handling, which must be modelled outside the faulty task element. The exception arc from the faulty task connects to a **catch** element, whose job is to issue event handling mechanisms depending on the type of the exception. Figure 16 shows possible handling scenarios. One catch node can branch into different scenarios depending on the exception type and other conditions.

4.1.2.3 Power and Energy model

In this work, we use power state modelling to describe system energy consumption. The total energy of system E is calculated as the time-weighted sum of powers:

$$E = \sum_i P_i \cdot t_i, \tag{4.1}$$

where P_i is the power dissipation during some state i, and t_i is the time spent in the state i. Ideally each task should correspond to its own power state as each task uses a different ratio of system resources. However, it is more practical to group power states by the task type, for instance, GUI tasks (Activity or Fragment), location service tasks, web service tasks, etc.

In addition to active power states associated with task execution, there are two idle power states: user idle state, when the application is displayed in the foreground and is awaiting for user input, and sleep state, when the application is closed, the screen is off, and the background services are awaiting their schedule.

The estimated duration of each state can be found from simulations, and the power dissipation can be found from the characterization experiments as described in Section 4.1.4.

4.1.3 Model Generation

This section presents the model generation from an Android application in order to perform the ArchOn simulation. To generate this model it is necessary a component and control flow analysis to identify elements of the application that need to be represented in the model and interactions between them.

4.1.3.1 Component Analysis

Subsection 4.1.2 shows modelling elements that represent a system. To simulate an application it is necessary to declare these elements in a SIM file and execute the ArchOn tool. Android applications have specific components (Activity, Service, AsyncTask, BroadCast Receiver, Fragment) to represent user interface, background services, and tasks. These components are implemented through Java classes and communicate with each other. Our implementation needs to analyzing these components and generate model elements. We used the Soot framework (VALLéE-RAI et al., 2000) and the Dexpler software package (BARTEL et al., 2012) to process Java bytecode of the application APK file.

In the first place, the implementation declares the package that contains the classes that represent the modeling elements and the Archon class that it is responsible for calculating the estimation values in the simulation. There are specific commands used in the SIM file declaration to declare modeling elements (assign) and declare the initial values of the properties of each element (setup), for example, the implementation has an internal list of all commands.

The first assignment generated is for the `User` that represents the person that will interact with the interface of the application and initiate some tasks. The User has properties necessary to the simulation like time of simulation, delay, power, battery, and tasks connected to it.

After that, the implementation iterates over the application classes in Java bytecode using an iterator provided by Soot element `Scene` to generate the assignments and setups. For each class inside the APK file, verification is run to check if the class is an application class, this is necessary because the file contains all classes of application, Android components, and third-party libraries. And if this class has some interaction with another component that will be reflected in the simulation. Another verification is made to verify if the class is an Android component. The implementation has a `HashMap` to save the classes that in the model will be a connection with a `Handler`, if the class has this connection the implementation write it like a `FaultyTask` in the SIM file, if not, the class will be a `Task` element. Tasks and FaultyTasks have properties of delay and power, but `FaultyTask` has properties to indicate the probability of an exception is thrown inside.

At last, the implementation generates assignments to Handlers, Invokers, and external components (remote server, other applications). The implementation has internal counters that save the number of interactions with handlers and invokers and based on this is generated the correct number of assignments and setups.

4.1.3.2 Control Flow Analysis

This analysis is done before the generation of assignments because the implementation needs the control flow information to identify the interactions between application components and save this information. The `User` has a maximum number of tasks that are defined by the designer, so the implementation will take this into account to generate the interaction. The `User` interacts with tasks that are user interface components inside the application like, Activities and Fragments. These tasks can start other tasks to perform background operations like AsyncTasks.

Interactions between tasks are modeled using request-acknowledge protocol 4.1.2. In the first step, the implementation analyzes the bytecode of ArchOn classes to get the name of the input and output ports of each modeling element. After that, the implementation iterates over application classes. For each class, the implementation iterates over the methods to analyze the method body and identify an interaction between the class and another application component. To iterate over units of a method it is used the Soot element `UnitGraph`.

In terms of bytecode analysis, interaction is a statement that contains an invoke expression that invokes another component of the application. For example, to invoke a `Service`, a component needs to declare an `Intent` object with the name of the calling class and the name of the class that represents the service and then calling the `startService` method passing the Intent object. To invoke an `AsyncTask` the class needs to call the `execute` method.

The implementation analyzes the body of each method of an application class identifying these types of interaction in the bytecode and writing this interaction in the SIM file using the information about input and output ports and information obtained from invocation expressions, classes, and methods.

4.1.4 Model Characterization Experiments

The main idea behind extra-functional simulation is that the model designer provides performance and power characteristics for individual components of the system, and the simulation then uses these characteristics whilst considering complex interactions between the components to provide estimates of the total performance and energy for the entire system.

Experimental power characterization of model components is a challenging task that may require special instrumentation in both hardware and software. The presented workflow does not depend on a specific characterization method. The method described in this section presents a good balance between the accuracy and effort. Section 6.4 in the chapter of related works describes a number of existing alternative techniques.

4.1.4.1 Power Characterization

Different platforms usually display different power characteristics, hence the absolute power values obtained on one platform cannot be applied globally. However, it is reasonable to assume that the relative power consumption of application components should stay approximately the same in all Android devices that use similar Java VM. Consequently, the relative differences between extra-functional properties of application modes can also be valid across platforms.

The proposed method of measuring the power consumption on an Android device is based on observing the battery level, so it relies on the built-in sensors and does not require any additional hardware instrumentation. To get the battery information, we use `batterystats` tool from Android and visualize the data with Battery Historian tool.

The method also requires minimal instrumentation of the target application to enable running the application in a specific power state, which corresponds to the Task element being characterized. In order to achieve that, the task can be frequently rescheduled to emulate a continuous execution of a task over a period of an experiment (one hour) while the number of task invocations and the total CPU time of the task are being tracked. The duration of the experiment must be long enough to have a detectable impact on the battery level.

The respective power consumption is derived from the reported battery use for the application by converting from mAh to Watts as follows:

$$\mathit{TaskPower}\,(\mathrm{W}) = \frac{3.6 \cdot \mathit{BatteryUse}\,(\mathrm{mAh}) \cdot \mathit{Voltage}\,(\mathrm{V})}{\mathit{TotalTaskTime}\,(\mathrm{s})}.$$

Voltage is the battery voltage, which is also available from the Battery Historian report.

According to Section 4.1.2.3, our model distinguishes two idle power states: *sleep* and *user idle*. Sleep state is when the screen of the device is off and only background processes are running. For sleep state characterization, we leave the device idle for one hour with no background applications running except the system-related processes.

$$\mathit{SleepPower}\,(\mathrm{W}) = \frac{3.6 \cdot \mathit{BgBatteryUse}\,(\mathrm{mAh}) \cdot \mathit{Voltage}\,(\mathrm{V})}{\mathit{TotalTime}\,(\mathrm{s})},$$

where *BgBatteryUse* is the sum of the battery use for all background processes, and *TotalTime* is the duration of the experiment.

User idle state happens when the application is open and displayed on the screen, but awaiting user input. Screen battery use can be found in the battery usage report in Android system settings. The reports provides the total time the screen has been on, and the percentage of the battery used specifically by the screen. By knowing the total capacity of the battery from the device specification, it is possible to calculate the screen power and idle power:

$$\mathit{ScreenPower}\,(\mathrm{W}) = \frac{\mathit{ScreenBattery}\,(\%) \cdot \mathit{Capacity}\,(\mathrm{Wh})}{\mathit{ScreenTime}\,(\mathrm{h})},$$

$$\mathit{UserIdlePower}\,(\mathrm{W}) = \mathit{ScreenPower}\,(\mathrm{W}) + \mathit{SleepPower}\,(\mathrm{W})\,.$$

Since the proposed characterization method relies on the battery level measurements, it is efficient for large-scale analysis, but fails to reliably determine the power of small tasks (less than 2ms), so it is not recommended for detailed application modelling. Section 5.1.3 in the chapter 5 discusses the advantages and drawbacks of the method in the context of an actual use case example.

4.1.4.2 Exception Probabilities

The ArchOn simulation takes into consideration, in addition to battery consumption, the probabilities of exceptions in the application. For this study was developed an application that performs static analysis and calculates probabilities at three levels: application level, classes or components level, and method level. This application makes use of the Soot framework to analyze the Android Package (APK) file of the application and generate the data in a CSV format.

There are two abstractions that represent the classes and methods of an application: `ProbabilityClass` that has attributes like name, a list of methods, a Hashmap which saves the probabilities of each exception in the class, a method count, the general probability of the class and the total probability of the methods of the class. And

`ProbabilityMethod` that has attributes like name, list of all exceptions that the method can raise, a list of exceptions that are declared in the exceptional interface of the method, a Hashmap that saves the number of times that an exception can be raised by one unit or statement inside the method, a Hashmap which saves the probabilities of each exception in the method, the number of units inside the method and the general probability of the method.

To calculate the probabilities the application developed first iterates over the classes of the Android application to analyze the methods of each class. For each method, it gets the Soot `ExceptionalUnitGraph` that provides an iterator for the units of the method and exception information of these units. For each unit of a method, the application gets a set of exceptions (`ThrowableSet`) that can be raised by unit using the Soot element `UnitThrowAnalysis` that provides this information executing the method `mightThrow`. After that, a check is made to see if that unit is protect iterating through a collection of `ExceptionDest` objects provided by the exceptional graph and verifying if there is any catch that handles an exception type for this unit. If the unit is unprotected the component sets the values of the attributes of the `ProbabilityMethod` and sets the values for the attributes of the `ProbabilityClass`.

The application developed calculates a probability that an exception is raised inside an android application, it indicates that a specific type of exception has more chance to be raised and not handled during the execution of the application. For the method level, the application calculates the probability in the following way: the total number of exceptions that may be raised inside the method is divided by the size of the units of the method. This total number of exceptions is the sum of exceptions that are declared in the exceptional interface and the number of exceptions that can be raised by units inside the method. The probability of a class is the average of the probabilities of its methods and the probability of the entire application is average of the sum of class probabilities.

4.1.5 Model Analysis and Reiteration

Once the model is generated and characterized, it is simulated in the ArchOn tool for a fixed period of virtual time (model time). The reported results include the following estimates for extra-functional properties:

- the number of issued user commands and the average response time (ms);
- the total amount of energy (Joules) consumed during the simulated period, as well

as the per-component breakdown of energy consumption;

- the total number of exceptions thrown and per-component statistics on each exception type;
- the number of successful calls per component type, which can also be used to track exception handler invocations.

These data can be used to make design choices on which error handling strategies should be used in order to provide an improved balance between performance, energy, and reliability. The changes are then coded back to the application, from which a new iteration of the model is automatically generated. It is also possible to make a quick iteration by changing the model directly; however, doing it through the application code ensures that the application and the model always stay synchronized.

The design decisions depend on the application. As a general guideline, it is recommended to start optimization from the components that have the most impact on the energy consumption. Saving energy at the expense of reliability or user experience can be reserved for alternative operation modes and conditions like low battery or unavailable network signal. In order to make the methodology feasible to be used in practice, next section describes the new mechanism created to support the proposed methodology requirements.

4.2 A Novel Exception Handling Mechanism For Android

In order to support the proposed methodology, we propose a new exception handling mechanism for Android applications, named DroidEH. Section 4.2.1 describes the concepts that underpin the DroidEH mechanism. Section 4.2.2 illustrates how to use DroidEH to handle exceptions in Android applications. Finally, Section 4.2.3 describes implementation details of the proposed solution.

4.2.1 Conceptual Model

The DroidEH mechanism is grounded on two main approaches: the *EFlow* model (CACHO et al., 2008; CACHO; COTTENIER; GARCIA, 2008) and the concept of *Holistic Fault Tolerance*(GENSH et al., 2017a).

EFlow(CACHO et al., 2008; CACHO; COTTENIER; GARCIA, 2008) is a platform-independent model of exception handling whose major goal is to make exception flows explicit, safe, and understandable by means of *explicit exception channels* and *pluggable handlers*. An explicit exception channel (channel, for short) is an abstract duct through which exceptions flow from a raising site to a handling site. More precisely, an explicit exception channel E_{cc} is a 5-tuple consisting of: (1) a set of *exception types* `E`, (2) a set of *raising sites* `RS`; (3) a set of *handling sites* `HS`; (4) a set of *intermediate sites IS*; and (5) a function `EI` that specifies the channel?s exception interface.

Exception types, as the name indicates, are types that, at runtime, are instantiated to exceptions that flow through the channel. The raising sites are loci of computation where exceptions from `E` can be raised. The actual erroneous condition that must be detected to raise an exception depends on the semantics of the application and on the assumed failure model. For reasoning about the exception flow, the fault that cause an exception to be raised is not important, just the fact that the exception is raised. The handling sites of an explicit exception channel are loci of computation where exceptions from `E` are handled, potentially being re-raised or resulting in the raising of new exceptions. In languages such as Java, both raising and handling sites are methods, the program elements that throw and handle exceptions.

If an explicit exception channel has no associated handlers for one or more of the exceptions that flow through it, it is necessary to define its exception interface. The latter is a statically verifiable list of exceptions that a channel signals to its enclosing context, similarly to Java's throws clause. In EFlow model, the exception interface is defined as a function (Ex1 $\rightarrow$ Ex2) that translates exceptions flowing (`Ex1`) through the channel to exceptions signaled (`Ex2`) to the enclosing exception handling context (EHC). EHCs are regions in a program where the same exceptions are always treated in the same way.

Raising and handling sites are the two ends of an explicit exception channel. Handling sites can be potentially any node in the method call graph that results from concatenating all maximal chains of method invocations starting in elements from `HS` and ending in elements from `RS`. All the nodes in such graph that are neither handling nor raising sites are considered intermediate sites. Intermediate sites comprise the loci of computation through which an exception passes from the raising site on its way to the handling site. Intermediate sites in Java are methods that indicate in their interfaces the exceptions that they throw, i.e. exceptions are just propagated through them, without side effects to program behavior. Note that the notions of handling, raising, and intermediate site are purely conceptual and depend on the specification of the explicit exception channel. They are also inherently recursive. For example, an intermediate site of an explicit exception channel can be considered the raising site of another channel.

A pluggable handler is an exception handler that can be associated with arbitrary EHC, thus separating error handling code from normal code. A single pluggable handler can be associated, for example, with a method call in a class `DataPersistUtil`, two different method declarations in another class, `UserData`, and all methods in a third class `DownloadFile`. In this sense, they are an improvement over traditional notions of exception handler. Another difference is that a pluggable handler exists independently of the exception handling context (EHC) to which it is associated. Therefore, these handlers can be reused both within an application and across different applications.

Holistic fault tolerance (HFT for short) has been recently proposed (GENSH; ROMANOVSKY; YAKOVLEV, 2016; GENSH et al., 2017a) to support engineering of a cross-layer fault tolerance coordination to impose modularity to perform cross-cutting error detection and recovery for mobile many-core applications and to ensure improved system performance and power consumption. In the core of this approach is the architectural pattern (GENSH et al., 2017a) and modelling techniques (GENSH et al., 2017b) which allow developers to make optimal decisions about holistic involvement of system components in system level error detection and error recovery for each specific error.

The HFT architecture (GENSH et al., 2017a) has two main components: HFT *controller* and HFT *agents*. The HFT *controller* is the central element of the architecture. It coordinates system-wide FT strategies and distributes available computer resources among the application components. In addition, it reconfigures the application components if it detects that application can operate faster or more reliably. The HFT *agent* is a special auxiliary object assisting the HFT controller. Each HFT agent is responsible for monitoring certain non-functional feature, such as error handling in one or more application components. The HFT agent monitors and, if needed, intervenes in the control flow of critical functions in application components. Figure 17 illustrates the architectural view of the example depicted in Figure 5 by means of DroidEH concepts. For the sake of simplicity, connections between the functional components are omitted. This architecture comprises four application components that are monitored by one or more HFT agents and provide the interface for the HFT controller. Thus, the given components can be used for monitoring their inner operation to reason about the state of the entire application. When an exception occurrence is detected by an HFT agent, the HFT controller is in-

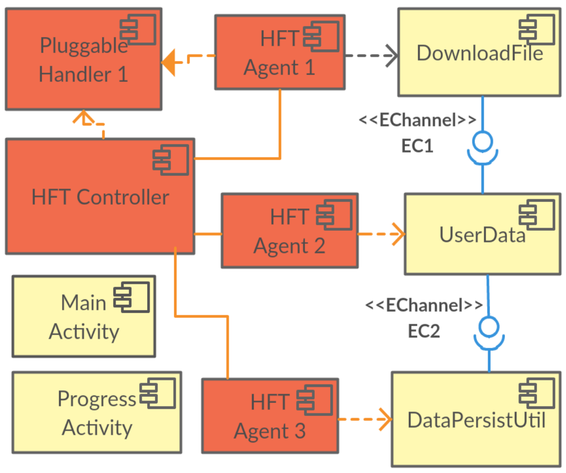


Figure 17: Architectural View of an Android Application using DroidEH. (OLIVEIRA et al., 2018a)

voked to check if there is any explicit exception channel defined for that exception. In that case, the HFT controller decides which pluggable handler should be used to handle that exception.

The main novelty of DroidEH is in supporting systematic engineering of holistic fault tolerance by applying cross-cutting reasoning about systems and their components. The EFlow model offers very useful and efficient structuring mechanisms for supporting cross-cutting thinking about complex systems. But it mainly leave with the developers the complexity of making the decisions about which system components to crosscut and how. The holistic fault tolerance tells the developers exactly which parts of the systems should be involved in tolerating the abnormal situations and how they should be involved. Hence the approach we are putting forward in this research work combines the benefits of these two complementary ways of architecting dependable systems.

4.2.2 DroidEH Abstractions

This section presents DroidEH abstractions, inherited from the EFlow model, expressed by means of Java annotations. DroidEH allows developers to define explicit exception channels and pluggable handlers in terms of the abstractions provided by Android.

4.2.2.1 Defining Raising Sites

DroidEH provides annotations for defining methods that can represent raising, intermediate or handling sites. This annotations are defined inside the class that represents the handler for the application. Raising sites are defined as follows:

```
@RaisingSites({
@AndroidRaisingSite(name = "rSite1",
target = "com.util.DataPersistUtil.getFileFromUrl")
})
```

Based on the hypothetical scenario in Figure 5, we define a raising site called `rSite1` bounded to method `DataPersistUtil.getFileFromUrl`. If the raising site spans for multiple classes and methods, DroidEH provides an economical way to define all those raising sites. For instance, one developer could use `com.util.DataPersistUtil.*` or `com.util.*` to define as the raising site all methods of `DataPersistUtil` and all method of package `com.util`, respectively.

4.2.2.2 Explicit Exception Channels

After the raising sites are defined, the developers define explicit exception channels using an annotation that takes 4 parameters: the channels name, the exception type, the raising site and the intermediate sites. Explicit channels are defined as follows:

```
@EChannels({
@AndroidExceptionChannel(name = "EC1",
exceptions =
"java.lang.IndexOutOfBoundsException",
raisingSite = "rSite1",
intermediateSite = "")
})
```

The above example defines one explicit exception channel called `EC1`, which represents exception `IndexOutOfBoundsException` flowing out of raising site `rSite1`, defined earlier. Notice that the definition of intermediate sites is not mandatory.

The implementation of *EChannels* attempts to locate methods that raise the exception supplied as an argument and considers them raising sites. A method can only be considered a raising site if the act of raising the exception is not a consequence of another exception, neither an implicitly propagated one nor an exception raised by a handler. The analysis then proceeds upwards, through the method call graph, considering every method to be part of the explicit exception channel, either as intermediate or handling sites. For the example given in Figure 5, method `getFileFromUrl()` is identified as raising sites of `IndexOutOfBoundsException`, and `UserData.downloadfile()` and `doInBackground()` as either intermediate and handling site. In summary, `EC1` matches all calls through which exceptions that were raised as a result of the execution of methods `getFileFromUrl()`, including calls to methods `doInBackground()` and `downloadfile()`.

It is important to note that though method `getFileFromUrl` is raising two exceptions, the `EC1` channel is only "conducting" exception `IndexOutOfBoundsException`. To capture all exceptions through the same channel, the following declaration of channel `EC1` could be used:

```
@EChannels({
@AndroidExceptionChannel(name = "EC1",
exceptions =
"java.lang.ArrayStoreException,
java.lang.IndexOutOfBoundsException",
raisingSite = "rSite1",
intermediateSite = "")
})
```

The problem of such exhaustive definition of exception types is that it can impair the usefulness of our approach. Hence, *EChannels* supports patterns to match exceptions related to a single class, a full class hierarchy, a class with a wildcard (*), or a combination of classes using logical operators. Therefore, rather than defining channels for exceptions `ArrayStoreException` and `IndexOutOfBoundsException`, one can use just `(java.lang.RuntimeException+)` to match all subtypes of `RuntimeException`. Note that the explicit need for adding of plus (+) after the exception type declaration eliminates the undesirable problems related to exception subsumption defined elsewhere (ROBILLARD; MURPHY, 2003). When the plus is not used, the exception subtypes are not captured by the channel.

The explicit exception channel defined above is too general once it will capture the signalling of any exception that is a subtype of `RuntimeException`. It is possible to specify more specific channels by explicitly indicating the raising sites of a channel. The code snippet below illustrates the definition of two explicit exception channels that include their respective raising sites:

```
@RaisingSites({
  @AndroidRaisingSite(name = "rSite1",
  target = "PersistenceDAO.*")
  @AndroidRaisingSite(name = "rSite2",
  target = "withincode(public void saveData())"
})
@EChannels({
@AndroidExceptionChannel(name = "EC1", exceptions = "java.lang.
    RuntimeException+", raisingSite = "rSite1", intermediateSite = ""),
  @AndroidExceptionChannel(name = "EC2", exceptions = "java.lang.
     NullPointerException", raisingSite = "rSite2", intermediateSite = "")
})
```

The second parameter of a channel definition identifies its raising site. The two examples above define the raising sites as separate annotations that the definitions of `EC1` and `EC2` use (Figure 5). The `EC1` definition captures all exceptions that are subtypes of `RuntimeException` and occur inside the model layer. The second channel `EC2` captures occurrences of one runtime exception (`NullPointerException`) raised within the body of method `saveData`. It follows that as long as the exception type is defined using the *EChannel* annotation, DroidEH does not make any distinction between checked and runtime exceptions. Both of them are equally monitored by the defined channels.

The *EChannel* annotation primarily supports the specification of channels that have a single raising site. Moreover, exception channels can also be associated with multiple raising sites as compositions of simpler channels, each containing a sole raising site. DroidEH supports the definition of multi-raising site channels by means Java's union operator:

```
@EChannels({
  @CompositeChannel(name = "ECCompose1",
  channels = "EC1 || EC2")
})
```

The use of the union operator allows us to keep simple the specification of both simple and complex channels and be coherent with Java's syntax and semantics.

A channel might fork at intermediate sites, resulting in two or more different exception control flows for the same exceptions. For example, suppose that there is an extra arrow from `getFileFromUrl` to `doInBackground` in Figure 5. If we wanted to define `EC1` to be exactly, it would be necessary to exclude this extra propagation "branch". The definition of a channel can be constrained even further. The programmer can include intermediate sites to the channel definition. In a similar vein, one can exclude some intermediate sites. In both cases, the semantics is to include or exclude the entire subtree of the channel whose root is the provided intermediate site. Intermediate sites, whether included or excluded, are supplied arguments to *Echannel.* The following snippet presents a simple example:

```
@RaisingSites({
```

```
@AndroidRaisingSite(name = "rSite1",
target = "com.util.DataPersistUtil.*")
})
@IntermediateSites({
@AndroidIntermediateSite(name = "iSite1",
target = "com.model.UserData.downloadfile")
})
@EChannels({
@AndroidExceptionChannel(name = "EC1",
exceptions =
"java.lang.RuntimeException+",
raisingSite = "rSite1",
intermediateSite = "iSite1"),
})
```

The *EChannels* annotation `EC1` above defines an explicit exception channel through which sub-types of exception `RuntimeException` flow. This channel has `rSite1` as its raising site. However, it only includes branches that crosses method `UserData.download-file` and ends at `doInBackground()`.

4.2.2.3 Pluggable Handler

Explicit exception channels defined with *EChannelsl* are incomplete, as they are associated with neither a handling site nor an exception interface. If one compiles a program that defines such a channel, the DroidEH will always indicate an error. For instance, if there are exceptions that should be propagated to the enclosing exception channel, but they are not part of the exception interface of the channel, then the program will not successfully be compiled. Unlike Java, DroidEH verifies if exceptions flowing through an explicit exception channel are handled or declared in its exception interface regardless of the exception type, i.e., these rules apply to both *checked* and *unchecked* exceptions.

In order to specify the handling site of an explicit exception channel, DroidEH provides the *AndroidExceptionHandler* annotation. This type of annotation supports the implementation of pluggable handlers. It encapsulates the exception handling code that is executed when a certain point in an explicit exception channel is reached. Each *AndroidExceptionHandler* annotation consists of: 1) the associated channel?s name; (2) the handling exception type; (3) the handling site, and (4) the body of the actual handler implementation that implements method `handler`. The following code snippet presents a simple handler:

```
public class HandlerExample
extends DroidEHHandler {

@AndroidExceptionHandler(
channel = "EC1",
catching = "",
handlingsite = "DownloadFile.doInBackground")
public void handler(EHContext context) {
Toast.makeText(context.getAppContext(),"Operation unsuccessful, please try
   again.", Toast.LENGTH_SHORT).show();
....
}
}
```

This method handles exceptions flowing through the channel `EC1`. The handler is activated when such exceptions reach method `DownloadFile.doInBackground`. The handler first display simple feedback about the operation in a small popup and invokes some methods to start again the `MainActivity` activity. For this hypothetical scenario, the user would have to choose another song. A pluggable handler can be associated with multiple explicit exception channels by means of Java's set union operator. The handler in the following code snippet is executed when exceptions from channels `EC1` or `EC2` are raised:

```
@AndroidExceptionHandler(
channel = "EC1 || EC2",
catching = "java.lang.SecurityException",
handlingsite = "com.activities.*")
public void handler(EHContext context) {
Log.d("Handling exception: ",
context.getException().toString() + "Exception in class " +
context.getDeclaringClass());
terminate(context.getException());
}
```

For the example above, the use of the *catching* clause can even further narrow the scope to which the handler is associated. This clause states that the handler should only be executed if the specified exception is caught. Hence, the example above catches any `SecurityException` flowing through EC1 or EC2 that reaches any method of the `activities` package. The defined handler first log the exception and then invokes the DroidEH's `terminate()` method. This method throws an exception to the Android runtime environment to crash the whole application.

4.2.2.4 Defining Channel Exception Interface

When an application cannot handle all the exceptions that flow through an explicit exception channel, it is necessary to declare these exceptions in the channel exception interface. The *eInterface* annotation serves this purpose. The following code illustrates typical examples of exception interfaces:

```
@eInterfaces({
@AndroidEinterfaces(channel = "EC1",
exception = "java.lang.NullPointerException+"
interfaceSite = "DataPersistUtil.*"
)
@AndroidEinterfaces(channel = "EC2",
exception = "",
interfaceSite = "DataPersistUtil.*"
)
})
```

The first declaration explicitly indicates that exception `NullPointerException` is part of the exception interface of the channel. Alternatively, the second declaration specifies only the explicit exception channel to which the exception interface is associated. This second format is more general. It states that every uncaught exception that flows through channel `EC2` is part of the channel exception interface.

Finally, the abstractions supported by DroidEH (channels, handlers, interfaces, etc.) enable holistic fault tolerance. They offer flexibility and modularity for introducing exception handling that crosscuts the components of any Android application, including activities, services, asynctask and broadcasts, as well as Java classes/objects. It is possible to clearly see where the explicit exception channels are defined and where exceptions are handled. In summary, the code snipped above shows that the DroidEH mechanism provides means to specify, in a local manner, non-local information pertaining to exception flows of an entire component or the software architecture implementation.

4.2.3 Implementation Details

The Figure 18 shows a overview of using DroidEH. To use the DroidEH abstractions in Android projects, the developer needs to import two libraries that provides the DroidEH annotation definitions: `DroidEHModel.jar` and `AnnotationDroidEHModel.jar`.

The `DroidEHModel.jar` library provides the base classes to create pluggable handlers (i.e.: the DroidEHHandler class). The `AnnotationDroidEHModel.jar` library provides the Java annotation definitions that allows us to specify the DroidEH abstractions as explained in section 4.2.2. The box `DroidEH Instrumentator` in the Figure 18 represents the DroidEH solution that has the following structure: codeinsert package, contains the main class that calls the transformer class and processes the annotation's data; model package, contains model classes for context, controller; transformers package, contains the class that instrument the bytecode of app and util package. The current implementation of DroidEH can be used in the Android Studio IDE after the following configuration:

1. Add the DroidEHModel.jar file as the library in the Android Studio project
2. Add the "annotationprocessor" module in the Studio project - File -> New -> Import Module
3. Add the lines below in "dependencies" in the build.gradle of the app module and synchronize: provided project (': annotationlibrary') annotationProcessor project (': annotationprocessor')
4. Create abstraction declaration class, for example, HandlerExample implementing the Handler interface
5. Declare channel annotations, raising sites, etc.
6. Give a build in the project to generate the APK and file with the notes data

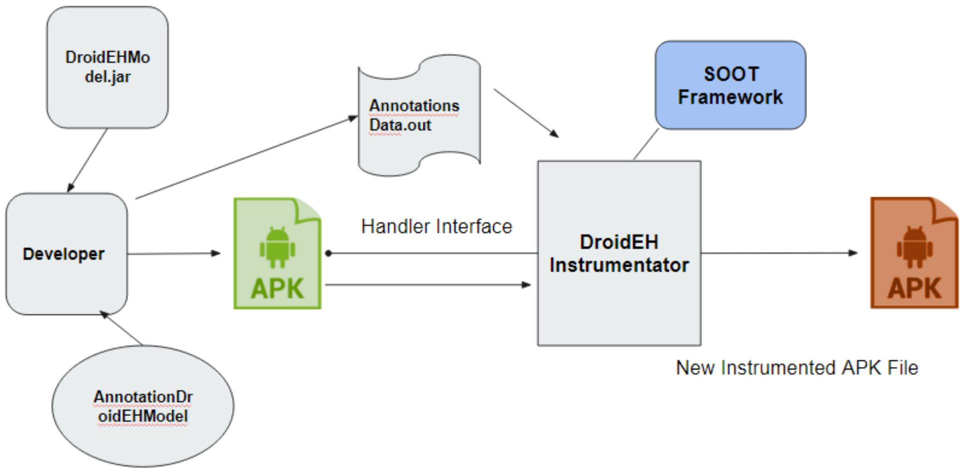


Figure 18: Overview of using DroidEH solution.

DroidEH receives the APK file of the app, package name of the app and data of annotations that the developer uses to describe the EH behaviour in terms of explicit exception channels, raising sites, interfaces and pluggable handlers that contains the implementation of EH strategies. The current implementation supports Error Handling agents and a controller to decide which handler call to handle an exception. The agents have access to the `context` object of the application and method information like declaring class, method name, exception interface of the method, exception caught message, etc.

DroidEH has a class called `MethodsTransformer`. This class performs two transformations: life cycle methods and raising sites. In the life cycle transformation, DroidEH inserts agents who call default handlers when an exception is detected, this transformation is performed in life cycle methods of the Android abstractions classes (Activity, Service, etc). The life cycle transformation is performed to mitigate problems pointed out in the study of the section 3.1. In the raising site transformation, the agents are inserted in the methods defined in the explicit exception channels. When an agent detects an exception, the controller is invoked to decide which pluggable handler should be used.

The Error Handling agents are implemented using Soot framework abstractions like "Trap" class that it's like a "catch" in bytecode level. And others Soot abstractions to insert the statements to execute the behaviour of the agent (call a handler, call the controller). The controller is a Java class (DroidEHHandler) that contains the implementation of default handlers.

After compiling the Android application, the Android Studio compiler provides as output an `AnnotationData.out` and an unsigned Android Package Kit (APK for short). The APK file format used by the Android operating system for distribution and installation of mobile applications. The `AnnotationData.out` contains information about (i) all declared explicit exception channels, (ii) a list of declared exception interfaces, and (iii) an internal representation of pluggable handlers which includes bodies, catching types, and channel attachment information. Based on those two files, we used the Soot framework (VALLéE-RAI et al., 2000) and the Dexpler software package (BARTEL et al., 2012) to process and instrument the Java bytecode of the unsigned APK file. This instrumentation allows to insert, by means of Soot transformation, the HFT agents and connect them to the HFT controller. At the end, we generate the instrumented APK and perform a post-instrumentation analysis to ensure that all exceptions flowing through the explicit exception channel are handled or declared.

To create explicit exception channels, we first obtain from the `AnnotationData.out` file the list of all declared explicit exception channels. Based on this list, we perform a Soot transformation to iterate through all the parts of a method where an exception can be raised. As soon as a raising site can occur, we construct, based on the algorithm proposed by Fu and Ryder (FU; RYDER, 2007), chains of exception-flow to link the corresponding raising site with its possible handling sites.

For each such handling site, the DroidEH checks each pluggable handler associated with that explicit exception channel in the application and determines if the handler's annotation can match that handling site. If it can match, we add a new entry to the enclosing method's exception handler table for the bytecode corresponding to the handling site. The code for the handler is *inlined* whenever the `handler` method body implements any return operation, otherwise the code is inserted as a call to the `handler` method. At the same time, checked exception(s) propagated along the call chain are declared in the method interface until meet some handler or channel's exception interface. At the end, we generate the instrumented APK and perform a post-instrumentation analysis to ensure that all exceptions flowing through the explicit exception channel are handled or declared. The implementation of method `retry` comprises bytecode instrumentation and the use of `ActivityManager`[2] component to get all tasks running in the current Activity and initiate the defined Activity.

DroidEH implements the EFlow model and inherits its abstractions to allow developers to define explicit exception channels and pluggable handlers. In a different way the implementation of DroidEH included specific handling methods for Android abstractions, such as method `terminate` that can be used by the agents or the controller, for example. In addition DroidEH implements HFT concepts like the HFT controller and the HFT agent.

[2]https://developer.android.com/reference/android/app/ActivityManager.html

5 Evaluation

This chapter presents the application of the proposed methodology in real Android applications and the evaluation of the proposed exception handling mechanism, DroidEH. The purpose is to assess the feasibility of using the proposed methodology to support developers to make decisions taking into account non-functional requirements, robustness and energy consumption, and to determine through the trade-off between these requirements, different operation modes that can be implemented in the application using the DroidEH. Moreover, this chapter assesses the adequacy of DroidEH for improving robustness.

5.1 Applying the Proposed Methodology: Use Case

This section describes our case study and some guidelines to employ our proposed methodology. The goal of this case study is to evaluate the impact of the proposed methodology on designing efficient mobile applications. Two versions of four target applications have been compared in order to observe the positive and negative effects of using our methodology. The first version (*V1*) was developed without using the proposed methodology and has one **Baseline** operation mode. In contrast, the alternative version (*VA*) was developed using the proposed methodology and has two operation modes: **DEH** and **DEH-LP**. DEH operation mode is represents the application that implements exception handling strategies, and DEH-LP operation mode is an operation mode in which the application applied some strategy for improve energy consumption, based in the execution of the methodology.

5.1.1 Target Applications

One major decision that had to be made for our investigation was the selection of the target applications. We have selected four applications with different purposes and functionalities. The first application is a medium-sized application (Visit Natal) to which there was a Java version available. The Visit Natal App is a mobile tourist guide application implemented in Java for Android Platform that aims to enhance tourists' travel experience. Visit Natal is the official tourist guide application for the city of Natal in Brazil and provides city visitors with a wide range of functionalities, such as the list of attractions to browse, detailed information about a specific attraction, how to arrive at an attraction, etc.

Fala Natal is an application for citizen engagement. The citizen can use the application to register a demand, for example, a broken pole on your street, register a compliment or a suggestion. To register a demand the citizen can use the phone camera to take a photo of the problem, choose an address and describe the demand. Also the citizen can consult the progress of the registrations. K9mail is an mail application where the user can manage various types of email accounts, the application supports three types: IMAP, POP3, and WebDAV. Using the application the user can view, send and receive e-mails for the registered accounts. Also the application send notifications for a new mail and error notifications. The last application selected is a Bitcoin Wallet, that is a bitcoin application where the user can send and receive bitcoins via NFC, QR codes or using bitcoin addresses, make payments via Bluetooth.

5.1.1.1 Interactions

The target applications have components that interact with each other to perform a given task according to user interaction. Identifying these interactions is important to generate a model of the application that can be used in the simulation and to guide the characterization of an application. Visit Natal has important scenarios related to its operation and main functionalities: Initialization of application, location, the update of data locally, show map routes, synchronization issues and send attraction reviews. The main scenario of Fala Natal is to register a demand, the application also allows viewing a specific demand or a list of demands and register a compliment or suggestion. For K9mail the main scenarios are send email and read email, and the main scenario for Bitcoin Wallet application is send bitcoins.

Each interaction related to the scenarios, has two or more components that interact by calling methods, sending and receiving messages. For example, for the location scenario in Visit Natal application, there are four components interacting, one activity, one fragment, one service and the web application. The activity initiates the fragment that shows the interface with the main functionalities of Visit Natal by calling the method `beginTransaction`. This fragment initiates a location service sending an Intent message to the Android platform, this service executes in the background sending GPS data of the user to the server making POST requests. This scenario is similar to the update of data locally, with the difference that in this scenario the service receives a message from the Web application with the data and saves in a local database.

In the send bitcoin scenario there are three interface components involved that represent the screens that the user needs to interact to finally execute the task. One component initiates other calling Android methods like `startActivity` and `setContentView`. These interface components interact with the blockchain service calling the method `start`. The register demand scenario in Fala Natal application also have interface components that interact with an asynctask component, calling the method `execute` and a remote server. In the scenario send mail for K9mail application the interface component interacts with an asynctask responsible for build and send the message.

5.1.1.2 Events and Handlers

The target applications have to handle in general input events and notification events. The input events are the result of user interaction with the UI elements on screen. Examples of notifications are when the application receives notifications from the server when the data changes, this is one type of notification event that needs to be handled, another type is generated inside the application when the location of the user is near to a specific tourist attraction, the application can receive a notification when receive a new mail.

For the input events like clicks, touches or focuses on the screen, and when the user selects a menu item, the application implements callback methods called event handlers, for example, to handle a click event, method `onClick` must be implemented inside an app. For the notifications from the remote server, the application implements a service class that has methods to receive and handle the notifications. From a notification generated based on user's location, the app calls a pending intent that shows a notification message on the phone.

5.1.2 Characterization Experiments

Section 4.1.3 in chapter 4, explains the methodology used to generate a model from the applications based on application's APK and app characterization. In order to do the application characterization it's necessary to execute characterization experiments to collect power and performance data from the components of the applications. This data is used in the model that will be simulated by ArchOn tool.

To execute the experiments we select a set of devices with different configurations in terms of model, Android version, battery capacity and memory. Table 13 shows the configurations. The objective of selecting different devices was to obtain relative values of energy consumption of the applications. The characterization of each application (5.1.1) was carried out through the selection of the main functionalities and construction of sequence diagrams for the components most likely to consume energy, describing their interactions with other components. Some of the application components are grouped together to reduce the model complexity. The result models consists of the following Task components for each application:

- **Visit Natal**: `LocationService`, service that runs in the background by continuously sending the user's location to the server in a given time interval and application data update service represented by `UpdateRatesIntentService`.
- **Fala Natal**: `NetworkAsyncTask`, asynctask that runs in the background, checking the user's location and network connection to show a map on the screen and `RegistrationRequestActivity`, uses the camera to capture photos that will be used to register the complaint in the application.
- **K9mail**: `onCheckMail`, checks for new messages to the user's inbox and `onOpen-Account`, loads the screen with all email information for a given user account.
- **Bitcoin Wallet**: `NetworkMonitorActivity`, monitor active peers and blocks to show the user that can click for more information on the internet and `SendCoins-Fragment`, allows the user to send coins using the camera or by address.

We also outline the following potential exception handling scenarios: retry failed task, add log entry, display error message, save state, terminate. All these scenarios, except retry, are modelled as special tasks and also have performance and power characteristics.

A total of 36 experiments were carried out according to the following run protocol: Select two components for each application, one hour of execution, screen on all the time. To minimize the contamination from other applications during the experiments we install only the target app on the device and use devices with only the Android system and proprietary apps running on the smartphone. Continuous execution of the

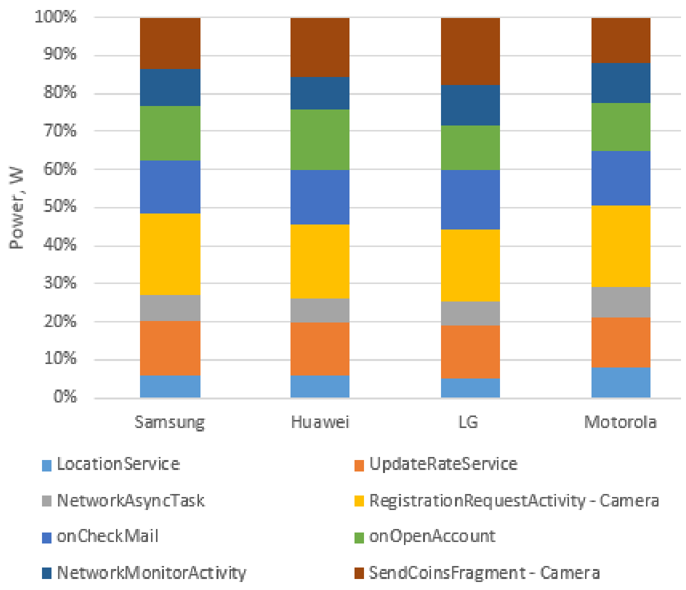


Figure 19: Energy consumption data of the components in different smartphones.

components, uninstall target app for the idle mode experiment, get the battery data using `batterystats` tool from Android and visualize the data with Battery Historian tool [1].

With the execution of the experiments, energy consumption data, such as voltage, battery level, percentage of battery usage by the target app and time execution were collected. With these data we can measure the energy consumption of each component of the applications during the experiments. Figure 19 shows the energy consumption of application components on different smartphones.

According to Figure 19, it can be identified that the Motorola device showed the most divergent energy consumption values, this device is the one with the oldest version of Android among the set of devices in the study. In order to find relative values of energy consumption we normalize the energy data and obtained the relative values of standard deviation of the Table 14.

With these values we can say that it is possible to carry out the characterization experiments by choosing only one of the devices, since the energy consumption of the components will be relatively similar in other devices. In this way, the experimental effort to obtain these values is reduced and the model used in the simulations can use the power data from one smartphone in the ArchOn tool. The developer has the freedom to choose to perform the experiments on more than one smartphone if he wants to evaluate his application in a specific configuration. In this case the SIM model needs to be generated

[1]https://bathist.ef.lc/

Table 13: Smartphones Configurations

Model	**Android Version**	**Battery Capacity**	**Memory**	**Voltage**
Samsung Galaxy S7 edge	8	3600 mAh	32GB 4GB RAM	3,85
Nexus 6P HUAWEI ce0168	6	3450 mAh	64GB 3GB RAM	3,82
LG Nexus 5X H791	6.0.1	2700 mAh	32GB 2GB RAM	3,8
Motorola Moto G XT1032	5.1	2070 mAh	8GB 1GB RAM	3,8

Table 14: Relative standard deviation values for application components.

Application Component	Relative std values (%)
LocationService	22,04
UpdateRateService	5,52
NetworkAsyncTask	13,65
RegistrationRequestActivity	7,47
onCheckMail	5,01
onOpenAccount	12,04
NetworkMonitorActivity	9,86
SendCoinsFragment	16,77

for each device.

5.1.3 Characterization Results

Table 15 presents the successfully measured component characteristics. The selected application components make use of some device components like internet, location, and camera. The `UpdateRatesIntentService` consumes more power than the `Location-Service`, the first component use the internet to get the rate value and saves the data in a local database, the `LocationService` get the location and sends to a remote service. The components that use the camera consumed more power than other in the same application, for example, `RegistrationRequestActivity` and `NetworkAsyncTask` in the Fala Natal application.

Any task that completes one request in a very short time, does not leave a distinguishable footprint in the battery use unless it is running continuously, but that would imply a complete redesign of the application, which we want to avoid in the proposed methodology. Our suggestion is to group small tasks into larger ones whenever possible or otherwise assume they have zero power and zero delay. Such zero-cost tasks are called

Table 15: Task power and performance characteristics

task	**delay, ms**	**power**, W
LocationService	10	1.108
UpdateRatesIntentService	10	2.772
NetworkAsyncTask	10	1.2474
RegistrationRequestActivity	2000	4.1580
onCheckMail	6	2.6334
onOpenAccount	100	2.7720
NetworkMonitorActivity	1000	1.8018
SendCoinsFragment	1000	2.6334
idle	—	0.554
sleep	—	0.554

token tasks, they do not have direct impact on system's extra-functional properties, but can still be counted during the simulations. For our use case, this is an accepted trade-off between model accuracy and design effort. In the target applications, exception handlers (logging, saving state, etc.) are considered as token tasks.

Table 16 shows the relative probability values of throwing a specific exception for each task. Exceptions types that are not considered specifically are grouped under the general `Throwable` interface. These values are provided by the static analysis of the application code as described in Section 4.1.4.2 of chapter 4. The absolute values are specifically large for the sake of fault injection.

5.1.4 Application of the Methodology

The first iteration of the model represents the original application behavior with no DroidEH infrastructure. The user interface components are missing handling of runtime exceptions listed in Table 16. For Visit Natal Location and data update background tasks are scheduled every 10ms and are always retried on failure. The components of the other applications are executed with the intervals shown in the Table 15.

Figure 20 shows the simulation-estimated active energy consumption per one hour of application usage; the original application behavior is called the **Baseline** mode. Since the model has stochastic elements (user events and exceptions), the presented result is averaged over one hundred simulations. It is worth noting that, for the model of this size, the ArchOn tool takes about 1–1.03s of real time to simulate 100 hours of virtual time on Intel Core i5-7200U at 2.7 GHz. The tool can handle larger and more detailed models; the bottleneck is in being able to characterize smaller size model components.

Table 16: Relative exception probabilities

	LocationService	UpdateRateService	NetworkAsyncTask	RegistrationRequestActivity	onCheckMail	onOpenAccount	NetworkMonitorActivity	SendCoinsFragment
Throwable	0.21	0.38	0.13	0.31	0.28	—	0.19	0.36
NullPointerException	0.19	—	0.19	0.11	0.07	—	0.21	0.09
ClassCastException	0.02	0.01	0.07	0.02	0.01	—	0.03	0.02
NegativeArraySizeException	—	—	—	0.01	0.01	—	0.01	—
ArrayIndexOutOfBoundsException	—	—	—	0.01	0.01	—	0.01	0.01
ArrayStoreException	—	—	—	—	0.01	—	—	—
MessagingException	—	—	—	—	0.01	—	—	—

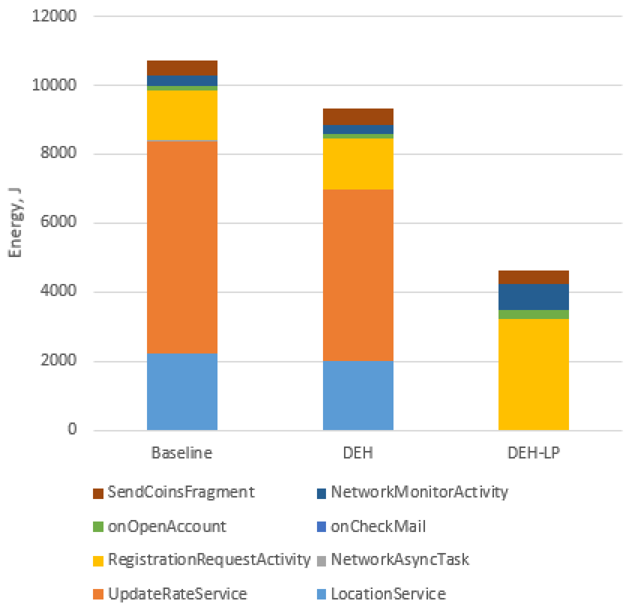


Figure 20: Per-component estimated energy in different application modes.

The simulation results reveal that the major contributors to energy consumption are the background services, hence they are selected as the primary targets for energy optimization. Also, the components that make use of the camera consume more energy. The main concern is that these background service tasks are being retried on failure while it may, in fact, be unnecessary as they are rescheduled every 10ms anyway. Therefore, in the second iteration of the workflow cycle, the error handling strategy for background service tasks are changed to lightweight handlers (error logging). The exception handlers are implemented using DroidEH annotations.

Additionally, we add lightweight handlers to all user interface components: exceptions in `MessagingController` of K9mail application lead to the graceful termination of the app, exceptions in `RegistrationRequestActivity` of Fala Natal application display an error message to the user, and exceptions in `WalletActivity` of Bitcoin Wallet application are followed by saving the application state. This iteration is labeled as the **DEH** mode in Figure 20. It can be observed from the simulation results that this mode of operation can save energy while maintaining an accepted level of fault tolerance. The reported energy saving is 13%; however, note that this number is tied to the exception probabilities, which are artificially increased in this model.

The benefit of the HFT with DroidEH approach is that HFT controller can switch between different application modes in runtime by means of using pluggable handlers that can be plugged and unplugged according to the operation mode. Hence the proposed workflow does not require each iteration to be an absolute improvement over the previous one, and can be used as an exploration of different operating modes instead. One of the possible alternative modes for the target application is the energy saving mode.

From the simulation results, we can see that the DEH mode still has the room for improving energy consumption at the cost of the background services; Also some components that use camera or components that make internet requests with data can be improved for energy consumption. However, at this point, any further reduction may impact user experience. In the **DEH-LP** mode we increase the scheduling period for the location and data update services from 10ms to 30s in the Visit Natal Application. For Fala Natal application we remove the component that verifies the internet connection and location in the registration of a demand. In the k9mail application we change the component to limit number of displayed folders, and in the Bitcoin Wallet application we limit the number of peers and blocks to be consulted by the component `NetworkMonitorActivity` Simulation results show 50% energy saving compared to the DEH mode (this time, the value is independent of the exception probabilities), but the application has significantly larger delays in exchanging the data with the server. Therefore, this mode is used only when the device battery is low and energy saving becomes a priority.

The conclusion after three iterations of the workflow is to configure the HFT controller to use DEH as the primary operating mode for the applications and switch to the DEH-LP mode when the battery level is low.

5.2 Assessment of DroidEH

DroidEH is conceived as a means to improve robustness of Android applications that have to deal with exceptional situations. This section presents an in-depth study performed to assess the adequacy of DroidEH for improving robustness.

5.2.1 Study Setting

5.2.1.1 Goal Statement and Study Hypothesis

The goal of this evaluation is twofold. First, we evaluate the adequacy of DroidEH for improving robustness of Android applications. Second, we compare the results gathered for the DroidEH implementations with the Java and a third-party implementation. The third-party implementation used in this study was proposed by Choi and Chang (CHOI; CHANG, 2015). This approach provides a component-level exception handling to allow developers to build robust Android applications. Hereafter, we refer to this approach as *Android Component-level Exception Mechanism* (*ACEM*, for short).

The definition of our hypotheses uses the following metrics to be collected and analysed: IT (Implementation time), NU (Number of exceptions uncaught), NC (Number of Crashes), and ST (Startup Time). IT is the amount of time spent by developers realising the tasks. NU denotes the number of uncaught exceptions collected through static analysis. NC is the number of crashes the testing tool found after executing the refactored applications. We collected ST using the information provided by Android `ActivityManager` in the logcat of Android Studio. According to the Android documentation [2], ST is defined as the amount of time elapsed between launching the process and finishing drawing the corresponding activity on the screen.

Each of these metrics has three variations corresponding to Java (J), DroidEH (D),

[2]https://developer.android.com/topic/performance/vitals/launch-time

and ACEM(A). For example, there are the Number of Crashes metrics defined for the Java (NC_J), DroidEH (NC_D), and ACEM (NC_A). The definitions of the following hypotheses use these symbols. The main hypothesis is the *null* hypothesis that states there is no difference in using either Java, DroidEH or ACEM. There are four null hypotheses, one for each metric of interest. Alternative hypotheses are defined to be accepted when each of the corresponding null hypotheses is rejected.

- **Null hypothesis** ($H0_1$): IT using DroidEH is not different from the ones for Java and ACEM approaches.
- **Null hypothesis** ($H0_2$): NU for DroidEH is not different from NU for Java and ACEM approaches.
- **Null hypothesis** ($H0_3$): NC for DroidEH is not different from NC for Java and ACEM approaches.
- **Null hypothesis** ($H0_4$): ST for applications that use DroidEH is not different from ST for applications that use Java or ACEM approaches.

5.2.1.2 Subjects

Four participants took part in this study. All participants were last-year undergraduate students with one or more years of programming experience in Android and an age range of nineteen to twenty-three years. None of the participants had prior knowledge of the applications. Their level of experience provided a good match for this study as we were interested in assessing the ability of DroidEH adopters to correctly grasp and use the notion of exception channels and pluggable handlers.

5.2.1.3 Target Applications

For the study we select 10 different Android applications written in Java to evaluate our solution (see Table 18). Hereafter we call them "target applications". This set of applications is practically the same used in the study of Chapter 3, Section 3.2.1.2, except for the MigrateClinic application, which in the previous study was replaced by another one.

As we are comparing DroidEH with ACEM implementations, we tried to use the same benchmarks used to assess the ACEM approach elsewhere(CHOI; CHANG, 2015). However, we were able to effectively use 8 out of 9 applications as we could not use

Table 17: Subject Android Applications

App	TotalLOC	TotalFiles	Apk Size
AndroidSecurity	932	19	343
BluetoothChat	667	3	26
Cafe	1570	13	1408
Contacts	1319	12	32
MigrateClinic	805	12	50
Noteslist	902	6	55
MediaPlayer	438	5	149
Bitcoin-Wallet	16662	114	3244
K-9-master	53696	345	5371
FalaNatal	37250	399	3893

Table 18: Experimental Results, Exception Flows Data, Execution Data and Application Test Data

		Implementation Time			Number of Uncaught Exceptions			Number of Crashes			Startup			TotalLOC
Application		ACEM	DroidEH	Java	ACEM	DroidEH	Java	ACEM	DroidEH	Java	ACEM	DroidEH	Java	Java
AndroidSecurity		5.0	10.0	7.0	2.0	1.0	6.0	0.0	0.0	0.0	521.0	678.0	462.0	932.0
BluetoothChat		16.0	16.0	13.0	1.0	0.0	16.0	0.0	0.0	59.0	209.0	355.0	366.0	667.0
Cafe		3.0	18.0	18.0	11.0	1.0	11.0	0.0	0.0	0.0	576.0	679.0	468.0	1,570.0
Contacts		7.0	22.0	23.0	9.0	3.0	14.0	2.0	0.0	0.0	493.0	471.0	522.0	1,319.0
MigrateClinic		9.0	11.0	10.0	5.0	0.0	7.0	0.0	0.0	60.0	0.0	0.0	0.0	805.0
NotesList		10.0	20.0	8.0	3.0	2.0	5.0	3.0	0.0	8.0	458.0	541.0	451.0	902.0
MediaPlayer		3.0	15.0	15.0	0.0	0.0	0.0	0.0	0.0	0.0	610.0	703.0	359.0	438.0
Bitcoin-Wallet		16.0	12.0	41.0	179.0	13.0	174.0	4.0	1.0	1.0	1,606.0	1,590.0	1,629.0	16,662.0
k-9-master		41.75	43.50	23.0	2,076.5	152.75	1,926.0	16.25	0.50	3.0	372.25	600.75	477.0	53,696.0
FalaNatal		19.5	18.75	17.0	43.0	3.0	44.0	1.25	0.0	5.0	704.25	821.50	620.0	37,250.0
Total	Mean	19.62	23.31	17.50	543.0	40.19	220.30	4.94	0.19	13.60	548.69	669.13	535.40	11,424.10
	Sum	314	373	175	8,688	643	2,203	79	3	136	8,779	10,706	5,354	114,241

the Earthquake application since it uses a deprecated version of Google Maps API. We replaced Earthquake app with two other applications: Fala Natal and K-9-Master. Hence, our set of applications consist of three commercial program (Bitcoin-Wallet, FalaNatal and K-9-Master), two sample programs (BluetoothChat and NotesList) developed by Google, one student project program (Cafe), and the remaining three programs excerpted from advanced Android applications development books (CHOI; CHANG, 2015). Table 17 shows the total LOC, number of files in the Android project, and the size of the APK file. The largest applications in term of LOC, number of files and size of the APK are Bitcoin-Wallet, K-9-master and FalaNatal.

We believe that these applications are representative of how exception handling is typically used to deal with errors in real Android software development for several reasons. First of all, these applications were selected mainly because they include a large number of exception handlers implementing very different exception handling strategies, from basic to sophisticated, including logging, return, throw, rollback, etc. Secondly, they have different characteristics, meet diverse requirements, represent different domains, and use a large set of real-world software technologies.

5.2.1.4 Study Phases

The study was divided into three major phases: (1) training, (2) developing the refactored versions of the target applications using the abstractions to be assessed (provided by Java, DroidEH and ACEM), and (3) assessing the set of versions developed in phase 2. The first phase comprised of an one-week training session in order to allow the participants to familiarize themselves with the exception handling abstractions provided by Java, DroidEH and ACEM. For this first phase we create a document showing an overview of basic concepts related to exception handling in the Java language, presenting their definitions and code examples. And showing exception handling in the Android perspective. In addition, the document shows the concepts and examples of using DroidEH and ACEM. During the training session, the subjects used a list of steps and guidelines to refactor a mobile application using Java, DroidEH and ACEM abstractions.

For the second phase, we designed the experiment as a *crossover design* (KUEHL, 2000), due to two reasons: (i) it requires fewer participants and (ii) it accounts for differences between participants. Crossover is a particular type of design where each experimental subject applies all treatments, but different subjects apply treatments in a different order. The use of a crossover design poses two well-known threats(VEGAS; APA; JURISTO, 2016): the effect of *order* and *carryover* from one technique to another. To reduce the effect of order, we defined two sequences with three periods (S1: DroidEH-ACEM-JAVA, S2: ACEM-DroidEH-JAVA). Each subject was randomly assigned to one of those sequences. At the end, two students performed S1 and two performed S2. For the carryover threat, we used the "two-stage procedure" to examine the possibility of carryover having occurred. We judged it has not occurred, then we believe the results of the analysis are reliable.

For sequence S1 we only considered the set of 8 applications used in the ACEM benchmark. For sequence S2, all subjects refactored the two remaining applications (Fala Natal and K-9-Master). Notice that for the case of Java refactoring, the subjects were asked to improve the exceptional behavior of the pre-existing Java version.

Each refactoring session consisted of the following steps. First, the experimenter introduced to the subjects the overall goal and format of the experiment. A printed copy of the task descriptions was given to each subject. The main task assigned to them was to identify the scenarios in which exceptions are not being handled and to refactor these scenarios using the approaches, each task is related to one application and one technique (DroidEH, ACEM or Java). It was not defined a number of scenarios that should be refactored, each participant could freely define this amount according to each application. The experimenter then ensured that the subjects could use the programming environment to edit, compile and run the target applications.

The subjects were given 4 hours to perform each task. The experimenter was present during this period and available to answer questions about the tasks and the programming environment. At the end of each task, which was either when the subjects finished the task, or the end of the 4 hours, the subjects completed a questionnaire regarding the perceived usability of the approach used. In this questionnaire the subject fills personal information like, name, age, gender, course name or area of expertise and level of academic training. After that, the subject responds three questions about the experiment: What functionalities of the approach used were useful; What difficulties were encountered during refactoring and What characteristics do you think should be added to the approaches. Then, the generated source code and the questionnaire were collected. We have also recorded the time necessary for each subject to finish the task (i.e, the IT metric).

The goal of the third phase was to compare the robustness of DroidEH applications with the robustness of Java and ACEM implementations. In order to support a multi-dimensional data analysis, the assessment phase was further decomposed in two stages. The first stage examines the overall robustness by means of software testing technique. A full test set was built from scratch. In order to reduce test effort and avoid systematic bias during test creation, test cases were automatically generated with an adequate tool support. We used *Monkey*[3], a stress-test tool that runs on device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. Which is not the best approach to evaluating if the exception handling code is correct, but it is a very useful tool for identifying crashes.

We created three seed values for the pseudo-random number generator (700, 701 and 702). Each seed value generated the same sequence of events for all application versions (Java, DroidEH, and ACEM), we defined a number of 7000 events for each seed. Each application version was executed on an Android Galaxy S5 device. Every crash identified during the assessment phase was documented in a customised report form and reflected in the *Number of Crashes(NC)* metric.

The second stage of the assessment phase aims to complement the test cases by using static analysis. We followed the typical approach adopted in the community in which the uncaught flow is used as an indicative of robustness (ROBILLARD; MURPHY, 2003; FU; RYDER, 2007; COELHO et al., 2008; MAJI et al., 2012). Exception flow is a path in a program

[3]https://developer.android.com/studio/test/monkey.html

call graph that links the method where the exception is raised to the method where the exception is handled. If there is no handler for a specific exception, the exception flow starts in the method where the exception is raised and finishes at the program entrance point. In the context of our study we used the *Uncaught Flow* metric to support our analysis, as it is often used as an indicative of software robustness (ROBILLARD; MURPHY, 2003; FU; RYDER, 2007; COELHO et al., 2008; MAJI et al., 2012). *Uncaught Flow* counts the number of exception flows that leave the bounds of the system without being handled. Therefore, this is the key indicator of a potential fault in the exception handling behavior and, therefore, often represents lower robustness. Uncaught exceptions terminate the execution of a program, hence, the higher the number of uncaught exception flows is, the lower the system robustness is.

We employed an extended version of the eFlowMining (GARCIA; CACHO, 2011) tool to collect the robustness metrics. eFlowMining is a multi-language static analysis tool that uses the approach proposed by (FU; RYDER, 2007) to perform an inter-procedural and intra-procedural dataflow analysis. The extended version of eFlowMining uses the Soot framework (VALLéE-RAI et al., 2000) and the Dexpler software package (BARTEL et al., 2012) to collect the structure (call graph, methods, exceptions, etc.) of Java and Android applications, respectively. Then, the tool generates the exception flows and computes NU for all exceptions, explicitly thrown by the application or implicitly thrown (e.g.,thrown by library methods). In this study we assume that only one exception is thrown at a time — the same assumption considered in (FU; RYDER, 2007). Unlike (CABRAL; MARQUES, 2007; CACHO et al., 2014), we have taken into account all exception flow types, including ones originating from unchecked exceptions, since some recent works (KECHAGIA; SPINELLIS, 2014; FRASER; ARCURI, 2015; KIM et al., 2013; COELHO et al., 2015) have found that `NullPointerException`, `IllegalArgumentException` and `IllegalState-Exception` are among the most common reasons for Java and Android applications to crash.

5.2.2 Study Results

This section presents the results of our measurement. Table 18 shows the mean of the implementation time (in min), the number of Lines of Code (LOC), the number of uncaught exceptions, the startup time (in ms), and the number of crashes for all tasks performed by the subjects for each target application. The results are grouped in terms of the three analysed approaches (Java, ACEM and DroidEH). The last row, named

“Total" indicates the *sum* and *mean* for each analyzed variable. To illustrate the different aspects of each finding, we provide a selection of quotes from the questionnaire. To enable traceability, each subject has an identification which can be traced in our database using the following convention: S#. For instance, S1 corresponds to an answer provided by Subject 1.

5.2.2.1 Implementation Time

A careful analysis of the total(*sum*) implementation time (IT) measures (Table 18) determines that with the ACEM approach, the subjects performed the tasks 15% faster than the subjects using the DroidEH approach. In the ACEM implementation, the subjects focus on refactoring the target applications by applying some rules defined by ACEM designers(CHOI; CHANG, 2015), such as: (i) every class that extends `Activity` should be replaced to extend `ExceptionActivity`, all *onCreate*, *onClick*, and *onActivityResult* declared methods should be renamed as *OnCreate*, *OnClick*, and *OnActivityResult*, etc. This simple set of rules to apply the ACEM approach was reported by some subjects. For instance, *"The use of ACEM is easy, requiring only the renaming of methods and classes used."*[S3]. In contrast, to apply DroidEH the subjects needed to capture the architectural view of the exceptional behavior and model it by means of DroidEH abstractions. For instance, *"To use DroidEH, I need to know all the flow of exceptions and its use requires a detailed knowledge of the code."*[S1]. In order to assess the first hypotheses whether the results for the three groups (ACEM, DroidEH, and Java) are different, we tested if there was a statistically significant difference between the values of IT computed for each approach. The results of the Friedman‘s ANOVA test for *IT* (χ^2 (2)=1.50, p = 0.653) showed that there is no statistically significant difference between them. In this manner, *we retain the null hypothesis ($H0_1$)*

5.2.2.2 Number of Uncaught Exceptions

In terms of the uncaught exception flow, the eFlowMining (GARCIA; CACHO, 2011) tool found 2,203 and 8,688 uncaught exceptions in the Java and ACEM approaches, respectively. The DroidEH implementations are superior with only 643 uncaught exceptions found. To further understand the possible causes of such difference between DroidEH and the others two approaches, we analyzed all the uncaught exception control flows and observed that most of them are related to uncaught exceptions raised by the methods defined by the Android platform like `execute`, from AsyncTask component, `startActivity`, from Activity component, `startService` and `bindService`, from Service component. These flows are most related to exceptions `android.content.ActivityNotFound-Exception`, `java.lang.SecurityException` and `java.lang.IllegalState-Exception`. As the ACEM approach does not provide corresponding implementation for those methods (like the OnCreate for the onCreate), they became unprotected and can raise many exceptions flows that eventually can cause an application crash. In contrast, the result of the DroidEH implementation is related to the enforcement of exception channels introduced at compile time. Reliability checks enforce exception flows to be made from the raising sites to their handling sites. This avoids the problems of having entry points and listeners methods without any recovery action. When such a harmful scenario occurs in the DroidEH implementation, the compiler issues a message making the problem explicit for the developer. By running the statistical test, the results of the Friedman?s ANOVA test for *UC* (χ^2 (2)=8.00, p = 0.005) showed that there is a statistically significant difference between them. ACEM and Java programs showed higher median values (2132.00 and 2001.00) against 157.5 in DroidEH. Thus, *we reject the null hypothesis ($H0_2$).*

5.2.2.3 Number of Crashes

For the number of crashes, the DroidEH refactored versions had 3 crashes against 136 in Java, and 79 for ACEM approach. This represents a reduction of around 90% and is a result of the DroidEH ability to handle exception and terminate the application when an exception is unrecoverable. Most of the crashes in the refactored ACEM apps were related to exceptions `java.lang.ClassCastException`, `java.lang.Instantiation-Exception` and `java.lang.NullPointerException`. The first two exceptions occurred inside components of the ACEM library. The crashes in apps using only Java approach were related to problems with *null* references, illegal arguments and class cast exceptions. The three crashes of the DroidEH approach were related to unrecoverable issues. Based on that numbers, the result of the Friedman‘s ANOVA test for *NC* (χ^2 (2)=6.50, p = 0.039) showed that there is a statistically significant difference between them. Therefore, *we reject the null hypothesis ($H0_3$).*

5.2.2.4 Startup Time

The data in Table 18 shows that DroidEH implementations tend to be worse in ST, and this difference gets larger when compared to Java. In fact, the result of the Friedman?s ANOVA test for *ST* (χ^2 (2)=8.00, p = 0.018) showed that there is a statistically significant difference between them. The median for ST_D was 657.5ms whereas the median for ST_A and ST_J was 548.00 and 539.5, respectively. Therefore, *we reject the null hypothesis ($H0_4$).* The ST_D data show that DroidEH implementations increases the average startup time in 0.11s as DroidEH introduces agents to monitor one or more system components. To better understand the effects of the DroidEH agents on the application performance, we look more closely to the time needed to run the stress-test. On average, each Java, ACEM and DroidEH app took 300.43ms, 317.84ms, and 327.65ms, respectively, to run 21,000 events (3 * 7000 events). It represents a very small increase of 0.027s for the DroidEH applications when compared to Java ones and this difference is never noticeable.

5.2.2.5 Execution Time of Monkey Tests

Table 19 shows the mean of the execution time, in seconds, that Monkey tool took to complete the execution of the tests for each seed value for each target application. The results are grouped in terms of the three seed values (700, 701 and 702). The last row, named “Total" indicates the *sum* and *mean* for each analyzed variable. During the execution of the tests, Monkey generates 7000 interaction events with the application interface. The “Total" column indicates that on average Monkey took longer to run the tests on applications that use DroidEH than on applications that use ACEM. In fact, the result of the Friedman?s ANOVA test for ST (χ^2 (2)=9.39, p = 0.008) showed that there is a statistically significant difference between the three variables. For seed 700, the average for DroidEH was 96.88%, for ACEM 94%. For seed 701, DroidEH averaged 108.74% versus 92% of applications using ACEM. For seed 703 this pattern repeats itself. For all seed values, Java-only applications have slightly longer run-time values than those using ACEM and smaller than applications that use DroidEH, except for seed 700, where Java has an average of 102.05%. Looking at the lines in Table 19 we can see that in six cases Monkey could not test the application that uses ACEM and in two cases the same applies to applications that use DroidEH, this influenced the final values of average runtime.

5.2.2.6 Questionnaire Data

In this subsection we will describe the answers obtained from the questionnaire filled out by the four subjects during experiment. This answers are important to have a feedback of DroidEH usability and limitations. Another important thing to note are the improvements proposed by the subjects that can be used as future directions for implementing

Table 19: Execution Time of Monkey Tests

		TimeSeed700			**TimeSeed701**			**TimeSeed702**		
Application		ACEM	DroidEH	Java	ACEM	DroidEH	Java	ACEM	DroidEH	Java
AndroidSecurity		103.82	97,53	111.02	97.95	121.87	92.25	99.84	105.24	86.20
BluetoothChat		100.60	103.90	137.25	108.50	118.60	116.16	75.15	133.25	126.76
Cafe		78.73	88.90	56.75	61.66	96.54	57.62	68.95	103.77	64.58
Contacts		118.02	136.23	106.75	107.54	134.28	99.86	141.17	244.23	110.55
MigrateClinic		0.0	113.65	125.78	0.0	113.65	141.85	0.0	113.65	141.85
NotesList		0.0	68.74	82.66	0.0	85.96	89.18	0.0	107.65	88.82
MediaPlayer		88.95	86.22	81.83	89.35	59.12	68.52	113.55	67.92	79.91
Bitcoin-Wallet		110.83	0.0	117.32	98.82	92.10	97.13	105.56	0.0	103.80
k-9-master		117.35	103.88	96.62	110.10	114.03	67.16	123.71	120.33	86.05
FalaNatal		108.40	109.86	104.56	116.96	115.42	102.11	97.17	120.81	104.72
Total	**Mean**	94	96.88	102.05	92	108.74	93.18	92.98	115.02	99.42
	Sum	1504	1550.17	1020.54	1472.08	1739.96	931.84	1487.74	1840.33	994.24

DroidEH.

The first question of questionnaire is related to what functionalities of the approach used were useful. Subjects points out that “DroidEH allows exception handling to be done externally to the source code of the application, allowing a modularization of error handling functions and a broad view of where they can occur”. Another subject says that “in DroidEH the fact that we do not change the original code in practically anything to be able to use it prevents errors related to the adaptation of the project to use the tool”. A subject points out that “DroidEH approach allows the exception handling specification to be well detailed”. The individual implementation for each type of exception thrown in the functions can be very useful. We can see that the modularization of exception handling is seen as a good thing by the subjects and the global view of exceptions of the application it’s another good funcionality for the subjects.

The second question is related to difficulties encountered during refactoring. This data is important to know what we need to improve to benefit from the developer’s use of the tool. Subjects points out that the “exhaustive registration of exceptions inside DroidEH annotations makes the process slower than treating the exception in the code itself”. This can be improved by providing autocomplete functionality. One subject argues that “DroidEH requires a knowledge of the whole flow of exceptions and a detailed knowledge of the code base, making the library difficult to use in applications with a high number of classes or by users who are not well versed in the source code of the application”.

Another observation of the subjects was that “the process of describing the exception handling with DroidEH is quite repetitive, but it requires a lot of attention, since there is a lot of naming and, if they are inserted erroneously, they do not present any type of error in time compilation”. A subject said that “the amount of annotations can end up being huge depending on the application, the class for handling the exceptions can get huge and with a difficult reading".

While DroidEH allows you to make extremely detailed and targeted exception handling, it also allows you to define generic handlers for multiple exceptional channels that filter generic exceptions, which may be an option for developers who have little knowledge of the application or want to generalize treatment in some part of the code. It is also interesting that DroidEH offers standard handlers for some types of exceptions that can be reused by the developer.

The last question of the questionnaire asks the subjects to point out functionalities that could be added in the tool. The subjects suggested some improvements: A path that searches through the code base for throw-exceptions and automatically registers them as "exception sites", or a way to create self-completion for the information being entered by the programmer in the annotations , which are currently treated only as normal strings; The writing of the RaisingSites and the Channels is very repetitive and end up getting big, a new form of writing can be thought, for example the exceptions treated in the Channels can be specified in another space and associated with a name, diminishing the size of the Channels; The intermediate is still very limited, it should allow specification between different classes.

The suggestions are directly related to Handler class writing and annotation, which has proved to be a repetitive and tiring process at times and should be taken into account in the implementation of another version of DroidEH.

5.2.3 Threats to Validity

This section discusses threats to validity that can affect the results reported in this work.

Internal Validity: Threats to internal validity are mainly concerned with unknown factors that may have had an influence on the experimental results (WOHLIN et al., 2000). To reduce this threat, we have selected a benchmark used to assess ACEM approach.

Construct Validity: Threats to construct validity concern the relationship between the concepts and theories behind the experiment and what is measured and affected (WOHLIN et al., 2000). To reduce these threats, we use metrics that are widely employed to measure software robustness (ROBILLARD; MURPHY, 2003; FU; RYDER, 2007; COELHO et al., 2008; MAJI et al., 2012).

External Validity: External validity issues may arise from the fact that all the data is collected from 10 application that might not be representative of the industrial practice. However, the heterogeneity of these systems helps to reduce this risk. They are implemented in Java, which is a representative language of the Android ecosystem. Furthermore, the characteristics of the selected applications, when contrasted with the state of practice, represent a first step towards the generalization of the achieved results.

Reliability Validity: This threat concerns the possibility of replicating this study. The source code of the target applications is publicly available. The way our data was collected is described in detail in Section 5.2.1.4. Moreover, all the tools used are available to obtain the same data. Hence, all the details about this study are available elsewhere (OLIVEIRA et al., 2018b) to enable other researchers to control it.

6 Related Works

6.1 Exception Handling and Robustness Studies

The first exception handling mechanisms date back to the early sixties with LISP (GABRIEL; STEELE JR., 2008) and PL/I (GRONER, 1971). Since the early days of exception handling, researchers have studied its relationship with program robustness (PARNAS; WURGES, 1976; MELLIAR-SMITH; RANDELL, 1977; LISKOV; SNYDER, 1979). This early work focused on the design of new mechanisms aimed at promoting robustness and modularity (LISKOV; SNYDER, 1979), on principles guiding the design and use of exception handling to improve these quality attributes (PARNAS; WURGES, 1976), and on the design of integrated solutions for software fault tolerance that leverage exception handling (MELLIAR-SMITH; RANDELL, 1977). Although much research has been conducted to study different aspects of the design and use of exception handling since then, there was a notable scarcity of empirical studies in this area. Only in the last 15 years, with the wide availability of large-scale, complex open source software systems, such empirical studies have started to surface. A number of these studies analyse design issues of exception handling mechanisms and practices of software developers that affect program robustness.

Marinescu (MARINESCU, 2011) analyzed three Eclipse releases and discovered that classes that throw or handle exception have a higher defect-proneness than classes that neither throw nor handle exceptions. A follow-up study by the same author (MARINESCU, 2013) has shown that classes using exceptions in Eclipse are more complex than those not using exceptions. Moreover, classes that use well-known exception handling antipatterns (MCCUNE, 2006), such as general `catch` blocks and general `throws` clauses have a higher probability of exhibiting defects than classes that do not employ such idioms. These studies differ from ours from section 3.1 in a number of ways. First, they do not target Android applications and analyze a single application, whereas we have studied 16. Second, they employ bug reports as indicators of defect-proneness, whereas we employ potentially uncaught exceptions elicited by means of static analysis, similarly to previous work (ROBILLARD; MURPHY, 2003; FU; RYDER, 2007). Third, we identify change scenarios that are more likely to impact program robustness.

Sawadpong *et al.* (SAWADPONG; ALLEN; WILLIAMS, 2012) analyzed six major releases of Eclipse and compared the defect densities of exception handling code and normal code. They used scripts to extract exception handling information from the source code of applications and also analyzed the bug reports. Their findings show that the defect density of exceptional code for the subject applications is considerably higher than the overall defect density.Our studies are similar in the sense that we both investigate the impact of exception handling on software robustness. However, our study is more precise because it describes the recurring change scenarios where possible exception handling faults were introduced, whereas their study only shows that exception handling code is more fault-prone than normal code. Moreover, our studies also differ in: (i) the metric used to measure robustness: we used the metric of *Uncaught Exception Flows*, while they used the metric *Defect Density*; (ii) the scope of the study, since we analyzed 16 different apps and a total of 112 versions; and (iii) the target application platform since, as shown previously, Android apps make extensive use of constructs that are not available for standard Java applications.

Coelho *et al.* (COELHO et al., 2008) studied the use of exception handling and software robustness in the context of aspect-oriented programs. The authors analyzed three medium-sized programs implemented in AspectJ. Similarly to our approach, they assessed the subject applications during their evolution and used the Uncaught Exception Flows metric. Their study showed that the exception handling code is error-prone, since the number of uncaught exception flows increased during program evolution. Nonetheless, Coelho *et al.* did not employ change metrics and did not explore the relationship between maintenance tasks and software robustness. Moreover, they analyzed fewer systems (only 3), these systems were not Android apps, and did not include constructs for asynchronous execution.

Linares-Vasquez et. al. (LINARES-VáSQUEZ et al., 2013) investigated the relation between the success of Android applications (in terms of user ratings) and the change- and fault-proneness of the underlying APIs. They have computed bug fixes and changes in the interfaces, implementation and exception handling of 7.097 Android applications belonging to different domains. They found that APIs used by successful apps (high user ratings) are significantly less change- and fault-proneness than APIs used by unsuccessful apps. In terms of changes to the set of exceptions thrown by methods, the study did not observe any significant difference between different levels of rating. These studies complement ours. Furthermore we have analyzed exception flows that may cause unknown bugs, due to our focus on the code, instead of preexisting stack traces or bug fixes. Also, we elicited change scenarios that have a negative impact on software robustness.

Cacho et al. (CACHO et al., 2014) conducted an empirical study with the goal of understanding the relationship between changes in C# programs and their robustness. This study targeted C# applications, whereas the subjects of our study are Android apps. Previous work has shown that Java and C# applications differ in the ways they use exception handling (CABRAL; MARQUES, 2007). Also, as mentioned before, Android apps make extensive use of constructs for asynchronous execution, and we have investigated whether the use of these constructs impacts program reliability. This kind of investigation could not be conducted within the context of C# applications.

Shaw and colleagues (SHAW; SHAW; UMPHRESS, 2014) evaluated 10,740 apps from the Slide Me market, and analyzed them using quality related metrics. The authors downloaded the APK files and reverse-engineered using the apktool to extract the Dalvik bytecode, with resources and the manifest file. The purpose of the analysis was predict marketing rating based on quality metrics by verifying the correlation between rates and the metrics. Our work performs a static analysis using the APK files to extract exception handling and structural information of applications, but differently from this work, we do not reverse engineered the APK files, we use the static tool Dexpler to analyse applications. In this work the authors proposes size metrics, object oriented metrics and Android Specific metrics that directly influence the user's experience, all this metrics can be obtained from an automatic way. Differently from this work, we propose change metrics and robustness metrics. The change metrics are only obtained from manual analysis.

Bhattacharya et al. (BHATTACHARYA et al., 2013) performed an in-depth empirical study on bugs in 24 widely-used open-source Android apps from diverse categories such as communication, tools, and media. They sought to understand the nature of bugs and bug-fixing processes associated with smartphone platforms and apps. They defined several metrics to analyze the bug fixing process. They showed how differences in bug life-cycles can affect the bug fixing process and performed a study of Android security bugs. The authors found that, although contributor activity in these projects is generally high, developer involvement decreases in some projects; similarly, while bug report quality is high, bug triaging is still a problem. Furthermore, they observed that in Android apps security bug reports are of higher quality but take more time to fix than non-security bugs. This study analyzed 24 Android apps, one version per app, while in our study we analyzed seven versions of each app taking into account the evolution of the applications. The focus of our study is on the analysis of the source code of the applications and analysis of the exceptional flows that may cause unknown bugs. Our study did not focus on security bugs.

White et al. (WHITE et al., 2015) proposed an approach to automate the process of reproducing a bug by translating the call stack from a crash report into expressive steps to reproduce the bug and a kernel event trace that can be replayed on-demand. To evaluate the solution, they used six open-source Android apps infected with errors. Our study evaluates how changes in the source code can impact the robustness of the application during evolution and analyzed multiple versions of the 16 subject applications.

In a different vein, Bavota et al. (BAVOTA et al., 2015) conducted two case studies to investigate how the fault- and change proneness of APIs used by Android apps are related to their success estimated as the average rating provided by the users to those apps. In the first study the authors analysed 5,848 free Android apps from the Google Play Market and investigated how the ratings that an app had received correlated with the fault- and change-proneness of the APIs such app relied upon. They used a set of tools to do this automatically, firstly they built a crawler to downloading the apps of market and select apps that have at least 10 votes, then they used a tool to convert the APK file to JAR file and a tool to extracted references/calls to API classes. Furthermore, the authors have mining commits from developers to identify bug-fixing commits activities and finally to analyse changes affecting APIs. In the second study the authors surveyed 45 professional Android developers to assess (i) to what extent developers experienced problems when using APIs, and (ii) how much they felt these problems could be the cause for unfavourable user ratings. Differently of this study, we analyse the Dalvik bytecode directly from APK file. The authors justified the second study because they did not have visibility over the source code of such apps and of their issue trackers and it is difficult to provide a strong rationale and, possible, a cause-effect relationship for findings of first study. In our study we conduct a manual analysis of source-code to identify change scenarios between versions of apps and correlate change metrics with the others metrics.

6.2 Exception Handling Approaches

ACEM (CHOI; CHANG, 2015) is the most related approach to our one since they propose a component-level exception handling to allow developers to build robust Android applications. They extended Android components like `Activity`, `Service` and `BroadcastReceiver` in a manner that all life cycle methods are protected by a generic *try/catch* block for catching all exceptions and relaunching to other components. All these methods have `java.lang.Throwable` in its exception interface. This approach differs from ours in a number of ways. First, they replaced the `startActivity` method for `startActivityFor-`
`Result` in a way that all Activities should return a result in order to support the propagation of exceptions. This mimics the exception handling mechanism of the standard Java and inherits all its side effects. For instance, there is no holistic view of the exception handling code and no reliability checks for exception control flows. Second, to use this approach, the developer needs to change all Activity, Service and BroadCastReceiver classes to use the ones provided by the proposed solution. It is also necessary to change calls to `onClick`, `startActivity`, and `startActivityForResult` methods.

In contrast, DroidEH offers developers an approach that supports a holistic view of the exception handling behavior. There is no need to change the application code, it is enough to just implement a handler class and use annotations. Finally, DroidEH performs reliability checks to ensure that exceptions are being handled appropriately. We have compared those approaches (and included Java in this comparison) in Chapter 5.2 and the observed results showed that DroidEH led to fewer uncaught exceptions and crashes. This indicates that the use of DroidEH abstractions helps to improve program robustness.

The work of (BEZERRA et al., 2019) presents an empirical study of Inter-Component Exception Notification in Android Platform. They investigated how Android developers deal with the problem of exceptions raised in parts of the application that are not responsible for handling the error, checking if exception notification were used in projects and how they were used. This is an inherent problem of the Android platform architecture that this research also tries to mitigate with the use of DroidEH. This work aims to contribute with the implementation of better mechanisms for communicating exception notifications in Java-based Android applications. The methodology proposed for this research work is more comprehensive, as it addresses other issues related to other non-functional requirements, as well as proposing a solution to improve the robustness of Android applications.

DroidEH solution addresses this problem globally through the EFlow model, protecting the different entry points of the application.

The work of (XIE et al., 2020) presents an approach to automatically detect exception handling defects related to external resources in Android applications. The work aims to mitigate problems of robustness of applications by implanting exception control codes into the input application an running UI tests. They evaluated the approach in six different types of applications with stable operation. This work is similar in terms of the problem faced, but the approaches are different in the sense that this work focus on the detection of defects and the present research work focus on a more comprehensive approach and the handling of errors that may occur in the application.

Other approaches related to our are described in subsection 2.6.1. These approaches implements the Eflow model like DroidEH but are not focused in Android applications and in solving the problems that were reported in this work related to exception handling in Android platform. The implementation included specific handling methods for Android abstractions, such as method `terminate`, for example. In addition DroidEH implements HFT concepts like the HFT controller and the HFT agent.

6.3 Robustness of Android applications

Several studies (KECHAGIA; SPINELLIS, 2014; ZHANG; ELBAUM, 2012; COELHO et al., 2017, 2015; WU et al., 2017; CHOI; CHANG, 2015) analyze issues in implementing exception handling code and practices of software developers that affect program robustness in Android. For instance, Kechagia and Spinellis (KECHAGIA; SPINELLIS, 2014) examine crash stack traces from 1800 Android applications. Their main finding is that most crashes originated from unchecked exceptions. More recently, Coelho and colleagues (COELHO et al., 2015) analyzed 6005 stack traces extracted from issues reported for 639 Android open source projects. They discovered that unexpected cross-type exception wrapping, e.g., catching an `OutOfMemoryError` and throwing an `Exception` that wraps it, and the use of undocumented exceptions, both checked (for the standard Android library) and unchecked (for both the Android library and third-party code), can be considered bug hazards (BINDER, 2000), circumstances that increase the chance of a bug to be present in the software. Zhang and Elbaum (ZHANG; ELBAUM, 2012) studied bug reports of five popular open source applications for the Android phone platform. By searching for "exception", "throw", and "catch" in the bug tracker and manually reviewing the results, the authors have identified 282 bug reports. Almost a third of the bugs that led to code fixes have been recognized as being caused by poor implementation of exception handling constructs. The authors also proposed an approach for amplifying existing tests to validate exception handling code associated with external resources. This work targets exceptions stemming from accesses to external resources, whereas our study does not make this distinction. Wu and colleagues (WU et al., 2017) analysed the Android source code and perceived that exceptions not caught by any application handler (or entry point handlers), will finally be trapped by function *uncaughtException* which is part of the exception handling mechanism of the Android platform. This function kills the exceptional process straightforwardly regardless of the process' attribute (WU et al., 2017). They have designed a protection extension, named ACEM, an Android patch that re-catches the uncaughtException exception and avoids the critical system services to be killed exceptionally. In a different vein, Choi and Chang (CHOI; CHANG, 2015) develop a component-level exception handling to allow developers to build robust Android applications. None of these studies focus on analysing the trade-off between exception handling and energy efficiency/performance. Moreover, they do not provide an approach to engineering efficient system-wide exception handling code.

6.4 Energy Estimation and Modeling for Mobile Applications

Measuring energy consumption in mobile application with power tools is expensive, time-consuming and requires skills out of the domain of most practitioners (CRUZ; ABREU, 2019). To overcome these issues, many approaches have been proposed to measure energy consumption using software-based estimators (Hao et al., 2013; CRUZ; ABREU, 2019). Hao and colleagues (Hao et al., 2013) propose a lightweight approach that provides fine-grained estimates of energy consumption at the code level. It achieves this using a combination of program analysis and per-instruction energy modeling. Le et al. (LE; BUI; TRUONG, 2019) proposed an approach to modeling and estimating power consumption based on the definition of power consumption automaton of hardware components. In this approach, developers can model source code with respect to power consumption and this can pinpoint which parts of source code might lead to energy leaks. Other researches observed that wrong choices made by programmers during the development tend to negatively influence the energy usage of mobile apps (PALOMBA et al., 2019). For instance, Sahin and colleagues (Sahin et al., 2012) highlighted the existence of design patterns that negatively impact the power efficiency, as well as the role of code obfuscation in the phenomenon (Sahin et al., 2014). Linares-Vasquez et al. (LINARES-VáSQUEZ et al., 2014) studied the API usage of Android apps and their relationship with energetic characteristics of apps. More recently, paper (CHOWDHURY et al., 2018) investigates the impact of the logging on Android application, finding that although there is little to no energy impact when logging is enabled for most versions of the studied applications, about 79% (19/24) of the studied applications have at least one version that exhibit medium to large effect sizes in energy consumption when enabling and disabling logging. Different from the above approaches, this work builds the energy consumption models to explore factors of energy consumption prevalent in the execution of the exception handling behaviour.

7 Final Remarks

This work is focused on improving the exception handling for Android applications and to provide the design of fault tolerant for Android applications. These applications are widely used by users worldwide and provide the most diverse services and purposes making use of device components like GPS, Camera, network connection, and so on. Hence, these devices have different characteristics and resource capacity.

Android applications are mostly written in Java that have an EHM where the program can indicate that an error has occurred by throwing an exception and this exception can be handled locally in the code. If the exception is not handled, the application may crash. So Android applications written in Java inherit this exception handling mechanism. However Android applications are different from standard Java applications, these applications have abstractions that are not present in Java applications and represent everything, from the user interface on the device, to services that run in the background. In addition, Android applications have multiple entry points that can throw exceptions during application execution and can be activated asynchronously. In Java standard applications this does not happen, there is only one entry point for the program. Android applications also have a different form of communication between components, through Intent objects. The Java EHM was not developed with these specific differences in mind, making it as a solution that does not fully address the problems that need to be mitigated in the exception handling of Android applications.

The exception handling mechanism provided by the Android platform has introduced two main problems for application developers: (1) the *"Terminate ALL"* approach and (2) the lack of a *holistic view* on exceptional behavior. Also, the use of exception handling may impact on the energy consumption of Android applications, in this way the application can consume more battery during its operation and impair the use of the device.

Studies (COELHO et al., 2015, 2017; WU et al., 2017; OLIVEIRA et al., 2018) have shown that there are problems related to the robustness of Android apps and lack of robustness from users' perspective (KHALID et al., 2015; MAN et al., 2016; LIM et al., 2015). Battery consumption is among the main complaints of users (KHALID et al., 2015; MAN et al., 2016), related to applications draining battery. This work evaluated the impact of exception handling on robustness and energy consumption of Android applications and proposed a general engineering methodology to help developers in making informed decisions about which architectures to select during the application development process. Through an interactive process the developers can make these decisions taking into account non-functional requirements such as robustness and energy consumption to build more efficient and reliable applications. A set of tools were developed and used to support this methodology, including the proposal for a new exception handling mechanism for Android applications.

The methodology was applied in real applications and proved to be satisfactory in achieving the objective of allowing the developer to make decisions taking into account these non-functional requirements and to determine through the trade-off between these requirements, different operation modes that can be implemented in the application using the DroidEH. The evaluation of DroidEH was also carried out in comparison to Java and a third-party implementation. The results showed that the use of DroidEH in applications of the study can enhance its robustness.

7.1 Main Contributions

To solve the problem presented in section 1.1, this work proposed a methodology to efficient engineering of Android applications and a new exception handling mechanism, called DroidEH, to support this methodology. Studies have also been carried out to better understand the impact of exception handling on the robustness and energy consumption in Android applications, and was carried out the evaluation of the proposed methodology and the proposed mechanism. The main contributions of this work are:

- Important findings from the first study to analyze the relationship between the usage of Android abstractions and uncaught exceptions: Increases in the number of uses of Android-specific abstractions are not in general related to increases in the number of lines of exception handling code; An increase in the use of Android-specific abstractions positively correlates (with a p-value of 0.01) with an increase in the number of uncaught exception flows; New exceptions being thrown without the accompanying `try` blocks; No implementation of the `UncaughtExeptionHandler` interface in the applications (OLIVEIRA et al., 2018).

- Study to investigate whether the use of exception handling strategies has an impact on energy consumption of Android applications. The results indicate that it is important to evaluate the energy consumption at exception handling level to make better EH design decisions.

- A comprehensive methodology that supports trading off power consumption, reliability and resource usage during exception handling design, a selection of tools and guidelines that support the methodology, including the automated model generation.

- Proposal and evaluation of a new exception handling mechanism for Android applications that uses the concepts of the EFlow model and HFT to improve the exception handling and robustness of Android apps (OLIVEIRA et al., 2018a; OLIVEIRA, 2018), that supports the proposed methodology.

7.2 Limitations

This work proposed a methodology for designing efficient Android apps and the EHM DroidEH. This methodology supports the developer or system designer in choosing appropriate HFT strategies for the application through an interactive process. The main limitation is that these tools that support the methodology are not integrated with DroidEH, and the model is not fully integrated with the application code.

Regarding the DroidEH, usability needs to be improved based on the questionnaire data applied on the study of chapter 5.2. To use the current version of DroidEH the developer needs to import the library and annotation module on Android Studio, could be interesting investigate a better integration with this IDE and with other IDEs like IntelliJ IDEA. The DroidEH was evaluated with Android applications written only in Java language, maybe it's interesting to evaluate the solution in Android applications written with Kotlin or generated by frameworks based in JavaScript, for example, to see if the solution needs any improvement.

Regarding to the experiments carried out, the energy consumption experiments used the `batterystats` tool from Android to get the battery data and visualize the data with Battery Historian tool. This is a good option because the visualization of the data is intuitive and the data collected is more accurate than using other types of Android profilers, but to have more accurate energy consumption data the experiments could use a hardware solution with a proper power meter. The energy consumption experiments of applications using some EH strategies gave an indication that can be interesting to evaluate the power consumption at that level in the application, the chosen strategies were simple to implement. It would be a good improvement to perform experiments with more applications and more complex EH strategies to expand the experiments.

7.3 Future Works

Future work could focus on the integration of the tools that support the proposed methodology, with this integration the changes in the model can be reflected in the application code more efficiently, including the changes that need to be done using DroidEH, like the instrumentation of the code based on the model behaviour or the addition of new explicit exception channels and handlers. Another future work could focus on improving the DroidEH implementation to mitigate the limitations of usability, integration with other IDEs, and different approaches to programming Android applications. Another option is migrate the solution to a web tool, with reporting, instrumentation, usage guides and tutorials. To better explore the proposed methodology, it can be applied in several applications from the critical domains, such as healthcare, law enforcement, well-being, transport and smart cities. As well as evaluating the use of the methodology by more developers who will be able to evaluate it. Another direction for future work can be to investigate further the trade-off between exception handling and energy consumption, doing experiments with EH strategies more complex and using a hardware solution with a proper power meter to collect energy data.

References

AL., A. F. G. et. A comparative study of exception handling mechanisms for building dependable object-oriented software. *The Journal of Systems and Software*, v. 59, n. 2, p. 197–222, 2001.

ALBAHARI, B. A. J. *C# 5.0 in a Nutshell: The Definitive Reference*. 5th. ed. [S.l.]: O'Reilly Media, 2012.

ARAÚJO, J. et al. Handling contract violations in java card using explict exception channels. In: IEEE. *2012 5th International Workshop on Exception Handling (WEH)*. [S.l.], 2012. p. 34–40.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 1, n. 1, p. 11–33, jan. 2004. ISSN 1545-5971. Disponível em: <http://dx.doi.org/10.1109/TDSC.2004.2>.

AYEWAH, N. et al. Using static analysis to find bugs. *Software, IEEE*, IEEE, v. 25, n. 5, p. 22–29, 2008.

BARBOSA, E. A.; GARCIA, A. Analyzing exceptional interfaces on evolving frameworks. In: *Fifth Latin-American Symposium on Dependable Computing Workshops*. [S.l.: s.n.], 2011. p. 17–20.

BARTEL, A. et al. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*. New York, NY, USA: ACM, 2012. (SOAP '12), p. 27–38. ISBN 978-1-4503-1490-9. Disponível em: <http://doi.acm.org/10.1145/2259051.2259056>.

BAVOTA, G. et al. The impact of api change-and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, IEEE, v. 41, n. 4, p. 384–407, 2015.

BEZERRA, V. L. et al. An empirical study on inter-component exception notification in android platform. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2019. p. 73–83.

BHATTACHARYA, P. et al. An empirical analysis of bug reports and bug fixing in open source android apps. In: IEEE. *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. [S.l.], 2013. p. 133–143.

BINDER, R. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. [S.l.]: Addison-Wesley Professional, 2000.

CABRAL, B.; MARQUES, P. Exception handling: A field study in java and .net. In: *21st European Conference Object-Oriented Programming*. [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 4609), p. 151–175.

CACHO, N. et al. How does exception handling behavior evolve? an exploratory study in java and c# applications. In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2014. p. 31–40.

CACHO, N.; COTTENIER, T.; GARCIA, A. Improving robustness of evolving exceptional behaviour in executable models. In: *Proceedings of the 4th International Workshop on Exception Handling*. New York, NY, USA: ACM, 2008. (WEH '08), p. 39–46. ISBN 978-1-60558-229-0. Disponível em: <http://doi.acm.org/10.1145/1454268.1454274>.

CACHO, N. et al. Ejflow: taming exceptional control flows in aspect-oriented programming. In: ACM. *AOSD ?08*. [S.l.], 2008. p. 72–83.

CACHO, N. et al. Trading robustness for maintainability: An empirical study of evolving c# programs. In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.]: ACM, 2014.

Castor Filho, F. et al. Exceptions and aspects: the devil is in the details. In: *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 2006. p. 152–162. ISBN 1-59593-468-5.

Castor Filho, F.; GARCIA, A.; RUBIRA, C. M. F. Extracting error handling to aspects: A cookbook. In: *IEEE International Conference on Software Maintenance, 2007. ICSM 2007*. [S.l.: s.n.], 2007. p. 134–143. ISBN 978-1-4244-1256-3.

CHIN, E. et al. Analyzing inter-application communication in android. In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. [S.l.: s.n.], 2011. (MobiSys '11), p. 239–252.

CHOI, K.; CHANG, B.-M. A lightweight approach to component-level exception mechanism for robust android apps. *Computer Languages, Systems & Structures*, Elsevier, v. 44, p. 283–298, 2015.

CHOWDHURY, S. et al. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering*, v. 23, n. 3, p. 1422–1456, Jun 2018. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-017-9545-x>.

COELHO, R. et al. Unveiling exception handling bug hazards in android based on github and google code issues. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2015. (MSR '15).

COELHO, R. et al. Exception handling bug hazards in android. *Empirical Software Engineering*, v. 22, n. 3, p. 1264–1304, Jun 2017. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-016-9443-7>.

COELHO, R. et al. Assessing the impact of aspects on exception flows: An exploratory study. In: VITEK, J. (Ed.). [S.l.]: Springer, 2008. (Lecture Notes in Computer Science, v. 5142), p. 207–234. ISBN 978-3-540-70591-8.

COTTENIER, T.; BERG, A. V. D.; ELRAD, T. Joinpoint inference from behavioral specification to implementation. In: SPRINGER. *European Conference on Object-Oriented Programming*. [S.l.], 2007. p. 476–500.

COTTENIER, T.; BERG, A. V. D.; ELRAD, T. The motorola weavr: Model weaving in a large industrial context. *Aspect-Oriented Software Development (AOSD), Vancouver, Canada*, v. 32, p. 44, 2007.

CRISTIAN, F. Exception handling and software fault tolerance. *IEEE Trans. Computers*, v. 31, n. 6, p. 531–540, 1982.

CRUZ, L.; ABREU, R. Emaas: Energy measurements as a service for mobile applications. In: *2019 41st International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2019.

CUI, X. et al. Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps. In: ACM. *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. [S.l.], 2015. p. 25.

EBERT, F.; CASTOR, F.; SEREBRENIK, A. An exploratory study on exception handling bugs in java programs. *Journal of Systems and Software*, v. 106, p. 82–101, 2015.

ERNST, M. D. Static and dynamic analysis: Synergy and duality. In: CITESEER. *WODA 2003: ICSE Workshop on Dynamic Analysis*. [S.l.], 2003. p. 24–27.

FILHO, F. N. d. P. *ECSFlow: implementação de um modelo de tratamento de exceção para C*. Dissertação (Mestrado) — Brasil, 2016.

FRASER, G.; ARCURI, A. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 20, n. 3, p. 611–639, jun. 2015. ISSN 1382-3256. Disponível em: <http://dx.doi.org/10.1007/s10664-013-9288-2>.

FU, C.; RYDER, B. G. Exception-chain analysis: Revealing exception handling architecture in java server applications. In: *ICSE '07*. [S.l.]: IEEE Computer Society, 2007. p. 230–239.

GABRIEL, R. P.; STEELE JR., G. L. A pattern of language evolution. In: *Celebrating the 50th Anniversary of Lisp*. [S.l.: s.n.], 2008. (LISP50), p. 1:1–1:10.

GARCIA, A. F. et al. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of systems and software*, Elsevier, v. 59, n. 2, p. 197–222, 2001.

GARCIA, I.; CACHO, N. eflowmining: An exception-flow analysis tool for .net applications. In: *Dependable Computing Workshops (LADCW), 2011 Fifth Latin-American Symposium on*. [S.l.: s.n.], 2011. p. 1–8.

GARTNER. *Market Share: Devices, All Countries, 4Q14 Update*. Mar 2015. Disponível em: <http://www.gartner.com/newsroom/id/2996817>.

GENSH, R. et al. Architecting holistic fault tolerance. In: *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. [S.l.: s.n.], 2017. p. 5–8. ISSN 1530-2059.

GENSH, R. et al. Modelling for systems with holistic fault tolerance. In: ROMANOVSKY, A.; TROUBITSYNA, E. A. (Ed.). *Software Engineering for Resilient Systems*. Cham: Springer International Publishing, 2017. p. 169–183. ISBN 978-3-319-65948-0.

GENSH, R.; ROMANOVSKY, A.; YAKOVLEV, A. On structuring holistic fault tolerance. In: *Proceedings of the 15th International Conference on Modularity*. New York, NY, USA: ACM, 2016. (MODULARITY 2016), p. 130–133. ISBN 978-1-4503-3995-7. Disponível em: <http://doi.acm.org/10.1145/2889443.2889458>.

GIGER, E.; PINZGER, M.; GALL, H. C. Comparing fine-grained source code changes and code churn for bug prediction. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2011. (MSR '11), p. 83–92. ISBN 978-1-4503-0574-7.

GOODENOUGH, J. B. Exception handling: issues and a proposed notation. *Commun. ACM*, ACM Press, New York, NY, USA, v. 18, n. 12, p. 683–696, 1975. ISSN 0001-0782.

GÖRANSSON, A. *Efficient Android Threading*. O'Reilly Media, 2014. ISBN 9781449364120. Disponível em: <https://books.google.co.uk/books?id=q_ivAQAACAAJ>.

GRONER, G. F. *PL/I Programming in Technological Applications*. [S.l.]: Books on Demand, Ann Arbor, MI, 1971.

Hao, S. et al. Estimating mobile application energy consumption using program analysis. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. p. 92–101. ISSN 0270-5257.

HAREL, D. Statecharts: A visual formalism for complex systems. *Science of computer programming*, Elsevier, v. 8, n. 3, p. 231–274, 1987.

HECHT, G.; MOHA, N.; ROUVOY, R. An empirical study of the performance impacts of android code smells. In: *Proceedings of the international conference on mobile software engineering and systems*. [S.l.: s.n.], 2016. p. 59–69.

HOPKINS, W. G. *A New View of Statistics*. nov. 2013. Disponível em: <http://www.sportsci.org/resource/stats>.

IDC. *Smartphone OS Market Share*. 2015. Disponível em: <http://www.idc.com/prodserv/smartphone-os-market-share.jsp>.

INC, G. *FY Q2 2013 Earnings Call*. jul. 2013.

INC., G. *The Android Developers Website*. 2015. Http://developer.android.com. Last visit: March 30th 2015.

JOHNSON, B. et al. Why don't software developers use static analysis tools to find bugs? In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 672–681. ISBN 978-1-4673-3076-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2486877>.

KECHAGIA, M. et al. The exception handling riddle: An empirical study on the android api. *Journal of Systems and Software*, Elsevier, v. 142, p. 248–270, 2018.

KECHAGIA, M.; SPINELLIS, D. Undocumented and unchecked: Exceptions that spell trouble. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014. (MSR 2014).

KERY, M. B.; GOUES, C. L.; MYERS, B. A. Examining programmer practices for locally handling exceptions. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2016. (MSR '16), p. 484–487. ISBN 978-1-4503-4186-8. Disponível em: <http://doi.acm.org/10.1145/2901739.2903497>.

KHALID, H. et al. What do mobile app users complain about? a study on free ios apps. *IEEE Software*, v. 32, n. 3, 2015.

KIM, S. et al. Predicting method crashes with bytecode operations. In: *Proceedings of the 6th India Software Engineering Conference*. New York, NY, USA: ACM, 2013. (ISEC '13), p. 3–12. ISBN 978-1-4503-1987-4. Disponível em: <http://doi.acm.org/10.1145/2442754.2442756>.

KUEHL, R. O. *Designs of experiments: statistical principles of research design and analysis*. [S.l.]: Duxbury Press, 2000.

LANDI, W. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 1, n. 4, p. 323–337, dez. 1992. ISSN 1057-4514. Disponível em: <http://doi.acm.org/10.1145/161494.161501>.

LE, H. A.; BUI, A. T.; TRUONG, N.-T. An approach to modeling and estimating power consumption of mobile applications. *Mobile Networks and Applications*, v. 24, n. 1, p. 124–133, Feb 2019. ISSN 1572-8153. Disponível em: <https://doi.org/10.1007/s11036-018-1138-4>.

LEE, J. K.; LEE, J. Y. Android programming techniques for improving performance. In: IEEE. *2011 3rd International Conference on Awareness Science and Technology (iCAST)*. [S.l.], 2011. p. 386–389.

LEE, P. A.; ANDERSON, T. *Fault Tolerance: Principles and Practice*. Second. [S.l.]: Springer-Verlag, 1990. (Dependable computing and fault-tolerant systems).

LI, D. et al. An empirical study of the energy consumption of android applications. In: IEEE. *2014 IEEE International Conference on Software Maintenance and Evolution*. [S.l.], 2014. p. 121–130.

LI, D. et al. Calculating source line level energy information for android applications. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2013. p. 78–89.

LIM, S. L. et al. Investigating country differences in mobile app user behavior and challenges for software engineering. *IEEE Transactions on Software Engineering*, v. 41, n. 1, p. 40–64, Jan 2015. ISSN 0098-5589.

LINARES-VáSQUEZ, M. et al. 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*. New York, NY, USA: ACM, 2013. (ESEC/FSE 2013), p. 477–487. ISBN 978-1-4503-2237-9. Disponível em: <http://doi.acm.org/10.1145/2491411.2491428>.

LINARES-VÁSQUEZ, M. et al. Mining energy-greedy api usage patterns in android apps: an empirical study. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014. p. 2–11.

LINARES-VáSQUEZ, M. et al. Mining energy-greedy api usage patterns in android apps: An empirical study. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014. (MSR 2014), p. 2–11. ISBN 978-1-4503-2863-0. Disponível em: <http://doi.acm.org/10.1145/2597073.2597085>.

LINARES-VASQUEZ, M. et al. How developers detect and fix performance bottlenecks in android apps. In: IEEE. *2015 IEEE international conference on software maintenance and evolution (ICSME)*. [S.l.], 2015. p. 352–361.

LIPPERT, M.; LOPES, C. V. A study on exception detection and handling using aspect-oriented programming. In: *ICSE '00*. New York, NY, USA: ACM Press, 2000. p. 418–427. ISBN 1-58113-206-9.

LISKOV, B.; SNYDER, A. Exception handling in clu. *IEEE Trans. Software Eng.*, v. 5, n. 6, p. 546–558, 1979.

LIU, Y.; XU, C.; CHEUNG, S.-C. Characterizing and detecting performance bugs for smartphone applications. In: *Proceedings of the 36th international conference on software engineering*. [S.l.: s.n.], 2014. p. 1013–1024.

MAJI, A. et al. An empirical study of the robustness of inter-component communication in android. In: *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. [S.l.: s.n.], 2012. p. 1–12. ISSN 1530-0889.

MAN, Y. et al. Experience report: Understanding cross-platform app issues from user reviews. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.: s.n.], 2016. p. 138–149.

MANOTAS, I. et al. An empirical study of practitioners' perspectives on green software engineering. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. (ICSE '16), p. 237–248. ISBN 978-1-4503-3900-1. Disponível em: <http://doi.acm.org/10.1145/2884781.2884810>.

MARINESCU, C. Are the classes that use exceptions defect prone? In: *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*. New York, NY, USA: ACM, 2011. (IWPSE-EVOL '11), p. 56–60. ISBN 978-1-4503-0848-9.

MARINESCU, C. Should we beware the exceptions? an empirical study on the eclipse project. In: *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*. [S.l.: s.n.], 2013. p. 250–257.

MCCUNE, T. *Exception-Handling Antipatterns*. 2006. Address: http://today.java.net/pub/a/today/2006/04/06/ exception-handling-antipatterns.html – Last access: May 10th 2013.

MELLIAR-SMITH, P. M.; RANDELL, B. Software reliability: The role of programmed exception handling. In: *Proceedings of an ACM conference on Language design for reliable software*. [S.l.: s.n.], 1977. p. 95–100.

MILLER, R.; TRIPATHI, A. Issues with exception handling in object-oriented systems. *Lecture Notes in Computer Science*, v. 1241, 1997. Disponível em: <citeseer.ist.psu.edu/297158.html>.

NAGAPPAN, N.; BALL, T. Use of relative code churn measures to predict system defect density. In: *Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005. (ICSE '05), p. 284–292. ISBN 1-58113-963-2.

NAKSHATRI, S.; HEGDE, M.; THANDRA, S. Analysis of exception handling patterns in java projects: An empirical study. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2016. (MSR '16), p. 500–503. ISBN 978-1-4503-4186-8. Disponível em: <http://doi.acm.org/10.1145/2901739.2903499>.

OKUR, S.; DIG, D. How do developers use parallel libraries. In: *Proceedings of the 21st ACM SIGSOFT Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2012.

OLIVEIRA, J. Taming exceptions in android applications. In: IEEE. *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. [S.l.], 2018. p. 174–177.

OLIVEIRA, J. et al. Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software*, v. 136, n. Supplement C, p. 1 – 18, 2018. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121217302558>.

OLIVEIRA, J. et al. An exploratory study of exception handling behavior in evolving android and java applications. In: *Proceedings of the 30th Brazilian Symposium on Software Engineering*. Maringa, Brazil: [s.n.], 2016. p. 23–32.

OLIVEIRA, J. et al. Droideh: An exception handling mechanism for android applications. In: IEEE. *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.], 2018. p. 200–211.

OLIVEIRA, J. et al. *https://sites.google.com/view/droideh-study/home*. May 2018.

OLIVEIRA, W.; OLIVEIRA, R.; CASTOR, F. A study on the energy consumption of android app development approaches. In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 42–52.

PALOMBA, F. et al. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, v. 105, p. 43 – 55, 2019. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584918301678>.

PARNAS, D. L.; WURGES, H. Response to undesired events in software systems. In: *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976. p. 437–446.

PHILLIPS, B.; HARDY, B. *Android Programming: The Big Nerd Ranch Guide*. 1st. ed. [S.l.]: Big Nerd Ranch Guides, 2013.

PINTO, G. et al. A large-scale study on the usage of java's concurrent programming constructs. *J. Syst. Softw.*, v. 106, n. C, p. 59–81, ago. 2015.

RAFIEV, A. et al. Studying the interplay of concurrency, performance, energy and reliability with ArchOn - an architecture-open resource-driven cross-layer modelling framework. In: *ACSD*. [S.l.]: IEEE Computer Society, 2014. p. 122–131.

RANDELL, B. The evolution of the recovery block concept. In: LYU (Ed.). *Software Fault Tolerance*. [S.l.], 1995. cap. 1, p. 1–21.

ROBILLARD, M. P.; MURPHY, G. C. Designing robust java programs with exceptions. In: *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2000. p. 2–10. ISBN 1-58113-205-0.

ROBILLARD, M. P.; MURPHY, G. C. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 12, n. 2, p. 191–221, 2003. ISSN 1049-331X.

ROBILLARD, M. P.; MURPHY, G. C. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 16, n. 1, p. 3, 2007. ISSN 1049-331X.

ROMANOVSKY, A.; SANDÉN, B. Except for exception handling &hellip;. *Ada Lett.*, ACM, New York, NY, USA, XXI, n. 3, p. 19–25, set. 2001. ISSN 1094-3641. Disponível em: <http://doi.acm.org/10.1145/568671.568678>.

Sahin, C. et al. Initial explorations on design pattern energy usage. In: *2012 First International Workshop on Green and Sustainable Software (GREENS)*. [S.l.: s.n.], 2012. p. 55–61.

Sahin, C. et al. How does code obfuscation impact energy usage? In: *2014 IEEE International Conference on Software Maintenance and Evolution*. [S.l.: s.n.], 2014. p. 131–140. ISSN 1063-6773.

SAWADPONG, P.; ALLEN, E.; WILLIAMS, B. Exception handling defects: An empirical study. In: *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*. [S.l.: s.n.], 2012. p. 90–97. ISSN 1530-2059.

SCHAEFER, C. F.; BUNDY, G. N. Static analysis of exception handling in ada. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 23, n. 10, p. 1157–1174, 1993. ISSN 0038-0644.

SENA, D. et al. Understanding the exception handling strategies of java libraries: An empirical study. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. [S.l.: s.n.], 2016. (MSR '16), p. 212–222.

SHAW, E.; SHAW, A.; UMPHRESS, D. Mining android apps to predict market ratings. In: IEEE. *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*. [S.l.], 2014. p. 166–167.

SHEN, H.; FANG, J.; ZHAO, J. Efindbugs: Effective error ranking for findbugs. In: *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2011. (ICST '11), p. 299–308. ISBN 978-0-7695-4342-0. Disponível em: <http://dx.doi.org/10.1109/ICST.2011.51>.

StatCounter Global Stats. *Android overtakes Windows for first time*. 2017.

STATISTA. *https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/*. Feb 2021.

VALLéE-RAI, R. et al. Optimizing java bytecode using the soot framework: Is it feasible? In: *CC '00: Proceedings of the 9th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2000. p. 18–34. ISBN 3-540-67263-X.

VEGAS, S.; APA, C.; JURISTO, N. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, IEEE, v. 42, n. 2, p. 120–135, 2016.

WHITE, M. et al. Generating reproducible and replayable bug reports from android application crashes. In: IEEE. *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*. [S.l.], 2015. p. 48–59.

Wilke, C. et al. Energy consumption and efficiency in mobile applications: A user feedback study. In: *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. [S.l.: s.n.], 2013. p. 134–141.

WOHLIN, C. et al. *Experimentation in Software Engineering - An Introduction*. [S.l.]: Springer, 2000. ISBN 978-0-7923-8682-7.

WU, J. et al. Exception beyond exception: Crashing android system by trapping in "uncaughtexception". In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. Piscataway, NJ, USA: IEEE Press, 2017. (ICSE-SEIP '17), p. 283–292. ISBN 978-1-5386-2717-4. Disponível em: <https://doi.org/10.1109/ICSE-SEIP.2017.12>.

XIE, L. et al. Automatically detecting exception handling defects in android applications. In: *12th Asia-Pacific Symposium on Internetware*. [S.l.: s.n.], 2020. p. 61–70.

YAU, S. S.; COLLOFELLO, J. S. Design stability measures for software maintenance. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 11, n. 9, p. 849–856, 1985. ISSN 0098-5589.

ZHANG, J.; MUSA, A.; LE, W. A comparison of energy bugs for smartphone platforms. In: IEEE. *2013 1st international workshop on the engineering of mobile-enabled systems (MOBS)*. [S.l.], 2013. p. 25–30.

ZHANG, P.; ELBAUM, S. Amplifying tests to validate exception handling code. In: *In Proceedings of 34th International Conference on Software Engineering*. [S.l.: s.n.], 2012. p. 595 –605.

APPENDIX A – Questionnaire form applied in the DroidEH experiment

Experimento - Informações

Nome:

Idade:

Gênero:

- Feminino
- Masculino

Nome do Curso/Área de aruação:

Nível:

- Graduação
- Mestrado
- Doutorado

Quais funcionalidades da abordagem utilizada foram úteis? (Explique o motivo para ambas).

Quais dificuldades foram encontradas durante a refatoração?

Quais características você acha que deveriam ser adicionadas nas abordagens? (Explique o motivo).

APPENDIX B – SIM file example for Visit Natal baseline version

```
#aliaspk "ncl.cs.prime.archon.arch.modules.tasks"
#estim ".TaskEstimation"

#assign User ".UserModelT" // UserModelT.TASKS = 0 Dummies: 1
#setup User "simulate: 360000; delayMean:5000; delaySDev:1000; idlePower
   :0.554; sleepPower:0.554; battery:10; pTask1:0.000"

// Component definitions and parameters
#assign UpdateRatesIntentService ".FaultyTask"
#setup UpdateRatesIntentService "type:UpdateRatesIntentService; preDelay
   :10; postDelay:0; power:2.772; pEx1:0.38; pEx3:0.01"

#assign LocationService ".FaultyTask"
#setup LocationService "type:LocationService; preDelay:10; postDelay:0;
   power:1.108; pEx1:0.21; pEx3:0.02; pEx2:0.19"

#assign Catch1 ".Catch"
#setup Catch1 "delay:0; power:0"

#assign MergeTask1Catch11 ".MergeAck"
#assign MergeTask1Catch12 ".MergeAck"
#assign MergeTask1Catch13 ".MergeAck"

#assign Catch2 ".Catch"
#setup Catch2 "delay:0; power:0"

#assign MergeTask2Catch21 ".MergeAck"
#assign MergeTask2Catch22 ".MergeAck"

#assign Timer1 ".Invoker"
#setup Timer1 "period: 10"
```

```
#assign Timer2 ".Invoker"
#setup Timer2 "period: 10"

#assign Retry1 ".Retry"
#assign Retry2 ".Retry"
// Interactions between Applications components

// ------- DUMMY -------
User.ack1 = User.req1

// Task->Invoker asynchronous

Catch1.ex = LocationService.ex
Retry1.req = Timer1.req
LocationService.req = Retry1.nextReq
Retry1.nextAck = LocationService.ack
Retry1.retry = Catch1.catch1
MergeTask1Catch12.ack1 = Retry1.ack
MergeTask1Catch12.ack2 = Catch1.catch3
MergeTask1Catch13.ack1 = MergeTask1Catch12.ack
MergeTask1Catch13.ack2 = Catch1.catch2
Timer1.ack = MergeTask1Catch13.ack
LocationService.nextAck = LocationService.nextReq
// Task->Invoker asynchronous

Catch2.ex = UpdateRatesIntentService.ex
Retry2.req = Timer2.req
UpdateRatesIntentService.req = Retry2.nextReq
Retry2.nextAck = UpdateRatesIntentService.ack
Retry2.retry = Catch2.catch1
MergeTask2Catch22.ack1 = Retry2.ack
MergeTask2Catch22.ack2 = Catch2.catch3
Timer2.ack = MergeTask2Catch22.ack
UpdateRatesIntentService.nextAck = UpdateRatesIntentService.nextReq
@loop
!
[^User.finished] #jump @loop
[^Timer1.finished] #jump @loop
[^Timer2.finished] #jump @loop
#estprint
!stop
```

APPENDIX C – JSON file generated by Archon tool

```
{
"Total time": 360003796,
"Total time (str)": "100h 0min 3s",
"User commands": 72117,
"Mean response time": 0,
"Total energy": 1038331,679,
"Total active energy": 838889,576,
"Total exceptions": 17113077,
"Simulation time": 68092,

"Energy per component": {
        "User (idle)": 199442,103,
        "LocationService": 222839,464,
        "Catch2": 0,000,
        "Catch1": 0,000,
        "User (sleep)": 0,000,
        "UpdateRatesIntentService": 616050,112
},

"Calls per task type": {
        "updateratesintentservice": 13554060,
        "locationservice": 11668758
},

"Exceptions per component": {
        "LocationService (ex3)": 401432,
        "LocationService (ex2)": 3817944,
        "LocationService (ex1)": 4223731,
        "UpdateRatesIntentService (ex3)": 221911,
        "UpdateRatesIntentService (ex1)": 8448059
}
}
```