Delft University of Technology
Software Engineering Research Group
Technical Report Series

Exception Handling Bug Hazards in Android: Results from a Mining Study and an Exploratory Survey

Roberta Coelho, Lucas Almeida, Georgios Gousios,
Arie van Deursen and Christoph Treude

Report TUD-SERG-2016-018

TU Delft

SERG

TUD-SERG-2016-018

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: This paper has been accepted for publication in *Empirical Software Engineering*, 2016, where it will have DOI http://dx.doi.org/10.1007/s10664-016-9443-7.

This paper is a substantially revised and extended version of Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie van Deursen: Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues. *Working Conference on Mining Software Repositories* (MSR) 2015: 134-145.

Noname manuscript No.
(will be inserted by the editor)

Exception Handling Bug Hazards in Android

Results from a Mining Study and an Exploratory Survey

Roberta Coelho · Lucas Almeida · Georgios Gousios · Arie van Deursen · Christoph Treude

Received: date / Accepted: date

Abstract Adequate handling of exceptions has proven difficult for many software engineers. Mobile app developers in particular, have to cope with compatibility, middleware, memory constraints, and battery restrictions. The goal of this paper is to obtain a thorough understanding of common exception handling *bug hazards* that app developers face. To that end, we first provide a detailed empirical study of over 6,000 Java exception stack traces we extracted from over 600 open source Android projects. Key insights from this study include common causes for system crashes, and common chains of wrappings between checked and unchecked exceptions. Furthermore, we provide a survey with 71 developers involved in at least one of the projects analyzed. The results corroborate the stack trace findings, and indicate that developers are unaware of frequently occurring undocumented exception handling behavior. Overall, the findings of our study call for tool support to help developers understand their own and third party exception handling and wrapping logic.

Roberta Coelho
Federal University of Rio Grande do Norte, CIVT/UFRN. Av. Senador Salgado Filho, 3000. Lagoa Nova, CEP: 59.078-970. Natal/RN. Brazil
Tel.: +55-84-3342-2216
E-mail: roberta@dimap.ufrn.br

Lucas Almeida
E-mail: lucas.almeida@ppgsc.ufrn.br

Georgios Gousios
E-mail: g.gousios@cs.ru.nl

Arie van Deursen
E-mail: arie.vandeursen@tudelft.nl

Christoph Treude
E-mail: christoph.treude@adelaide.edu.au

Keywords Exception handling · Android development · Repository mining · Exploratory survey

1 Introduction

The number of mobile apps is increasing at a daily rate. If on the one hand they extend phones' capabilities far beyond the basic calls, on the other hand they have to cope with an increasing number of exceptional conditions (e.g., faults in underlying middleware or hardware; compatibility issues [39]; memory and battery restrictions; noisy external resources [58]).

Therefore, mechanisms for exception detection and handling are not an optional add-on but a fundamental part of such apps. The exception handling mechanism [23], embedded in many mainstream programming languages, such as Java, C++ and C#, is one of the most used techniques for detecting and recovering from such exceptional conditions. In this paper we will be concerned with exception handling in Android apps, which reuses Java's exception handling model.

However, such mechanisms are more often than not the least understood and tested parts of the system [41, 45, 49, 21, 22, 13, 16, 57]. As a consequence they may inadvertently negatively affect the system: exception-related code may introduce failures such as uncaught exceptions [28, 58] – which can lead to system crashes, making the system even less robust [16].

In Java, when an application fails due to an uncaught exception, it automatically terminates, while the system prints a stack trace to the console, or to a log file [24]. A typical Java stack trace consists of the fully qualified name of the thrown exception and the ordered list of methods that were active on the call stack before the exception occurred [24, 11]. When available, the exception stack traces provide a useful source of information about system crashes [8] which can enable different kinds of post-mortem analysis and support: debugging [48], bug classification and clustering [54, 31, 18], automated bug fixing [51] and fault-proneness prediction models [32].

This work is conducted in two phases. First, a mining study performs a post mortem analysis of the exception stack traces included in issues reported on Android projects hosted on GitHub and Google Code. The goal of this study is to investigate whether the reported exception stack traces can reveal common *bug hazards* in the exception-related code. A *bug hazard* [10] is a circumstance that increases the chance of a bug being present in the software. An example of a bug hazard can be a characteristic of the exception-related code which can increase the likelihood of introducing the aforementioned uncaught exceptions. Second, we conducted an exploratory survey with the Android developers involved in the mined projects to assess their perspective about the exception handling bug hazards found.

To guide this investigation we compiled general guidelines on how to use Java exceptions proposed by Gosling [24], Wirfs-Brock [56] and Bloch [11]. Then, using a custom tool called ExceptionMiner, which we developed specif-

ically for this study, we mine stack traces from the issues reported in 482 Android projects hosted on GitHub and 157 projects hosted on Google Code. Overall 159,048 issues were analyzed and 6,005 stack traces were extracted from them. The exception stack trace analysis was augmented by means of bytecode and source code analysis for the exception-related code of the Android platform and Android applications. Some *bug hazards* consistently detected during this mining study include:

- Cross-type exception wrappings, such as an OutOfMemoryError wrapped in a checked exception. Trying to handle an instance of OutOfMemoryError "hidden" in a checked exception may bring the program to an unpredictable state. Such wrappings suggest that, when (mis)applied, the exception wrapping can make the exception-related code more complex and negatively impact the application robustness.
- Undocumented runtime exceptions raised by the Android platform (35 methods) and third-party libraries (44 methods) – which correspond to 4.4% of the reported exception stack traces. In the absence of the "exception specification" of third-party code, it is difficult or even infeasible for the developer to protect the code against "unforeseen" exceptions. Since in such cases the client usually does not have access to the source code, such undocumented exceptions may remain uncaught and lead to system crashes.
- Undocumented checked exceptions signaled by native C code. Some flows contained a checked exception signaled by native C code invoked by the Android Platform, yet this exception was not declared in the Java Native Interface invoking it. This can lead to uncaught exceptions that are difficult to debug.
- A multitude of programming mistakes – approximately 52% of the reported stack traces can be attributed to programming mistakes. In particular, 27.71% of all stack traces contained a java.lang.NullPointerException as their root cause.

The exploratory survey was conducted with 71 Android developers involved in one or more of the GitHub Android projects whose issues were mined. This survey reveals that only few developers (3% of respondents) knew about the undocumented checked exceptions signaled by native C code of the Android platform. Moreover, most of the developers recognized that cross-type exception wrappings may negatively impact the application robustness. The uncaught exceptions due to errors in programming logic, e.g. the NullPointerException, were identified by most developers (68%) as the first or second main cause of application crashes.

The high prevalence of NullPointerExceptions found in the mining study and confirmed in the exploratory survey is in line with findings of earlier research [32, 20, 17], as are the undocumented runtime exceptions signaled by the Android Platform [30].

Some of the findings of our mining study emphasize the impact of these bug hazards on the application robustness by mining a different source of in-

formation as the ones used in previous works. The present work mined issues created by developers on GitHub and Google Code, while previous research analyzed crash reports and automated test reports. Furthermore, our work points to bug hazards that were not detected by previous research (i.e., cross-type wrappings, undocumented checked exceptions and undocumented runtime exceptions thrown by third-party libraries) which represent new threats to application robustness. Moreover, we perform the first exploratory survey study whose goal was to assess developers' perspective regarding a set of exception handling bug hazards as well as how developers deal with exceptions and prevent crashes while developing.

Our findings point to threats not only to the development of robust Android apps, but also to the development of any robust Java-based system. Hence, the study results are relevant to Android and Java developers who may underestimate the effect of such *bug hazards* on the application robustness, and who have to face the difficulty of preventing them. Moreover, such *bug hazards* call for improvements of languages (e.g. to prevent null pointer dereferences) and tools to better support exception handling in Android and Java environments.

The remainder of this paper is organized as follows. Section 2 provides the necessary background on the Android platform and the Java exception model. Section 3 presents the mining study design, describes the ExceptionMiner tool we developed to conduct our study, and reports the study results. Section 4 details the exploratory survey. Section 5 provides a discussion of the wider implications of our results, presents the threats to validity associated with the mining study and discusses the limitations of the survey-based study, and points to the replication. Finally Section 6 describes related work, and Section 7 concludes the paper and outlines directions for future work.

2 Background

2.1 The Android Platform

Android is an open source platform for mobile devices based on the Linux kernel. Android also comprises (i) a set of native libraries written in C/C++ (e.g., WebKit, OpenGL, FreeType, SQLite, Media, C runtime library) to fulfill a wide range of functions including graphics drawing, SSL communication, SQLite database management, audio and video playback etc; (ii) a set of Java Core Libraries including a subset of the Java standard libraries and various wrappers to access the set of C/C++ native libraries using the Java Native Interface (JNI); (iii) the Dalvik runtime environment, which was specifically designed to deal with the resource constraints of a mobile device; and (iv) the Application Framework which provides higher-level APIs to the applications running on the platform.

2.2 Exception Model in Java

Exception Types. In Java, exceptions are represented according to a class hierarchy, in which every exception is an instance of the Throwable class, and can be of three kinds: the checked exceptions (extends Exception), the runtime exceptions (extends RuntimeException) and errors (extends Error) [24]. Checked exceptions received their name because they must be declared in the method's *exception interface* (i.e., the list of exceptions that a method might raise during its execution) and the compiler statically checks if appropriate handlers are provided within the system. Both runtime exceptions and errors are also known as "unchecked exceptions", as they do not need to be specified in the method *exception interface* and do not trigger any compile time checking.

By convention, instances of Error represent unrecoverable conditions which usually result from failures detected by the Java Virtual Machine due to resource limitations, such as OutOfMemoryError. Normally these cannot be handled inside the application. Instances of RuntimeException are implicitly thrown by the Java runtime environment when a program violates the semantic constraints of the Java programming language (e.g., out-of-bounds array index, divide-by-zero error, null pointer references). Some programming languages react to such errors by immediately terminating the program, while other languages, such as C++, let the program continue its execution in some situations such as the out-of-bounds array index. According to the Java Specification [24] programs are not expected to handle such runtime exceptions signaled by the runtime environment.

User-defined exceptions can be either checked or unchecked, by extending either Exception or RuntimeException. There is a long-lasting debate about the pros and cons of both approaches [3, 2, 1]. Section 2.3 presents a set of best practices related to each of them.

Exception Propagation. In Java, once an exception is thrown, the runtime environment looks for the nearest enclosing exception handler (Java's try-catch block), and unwinds the execution stack if necessary. This search for the handler on the invocation stack aims at increasing software reusability, since the invoker of an operation can handle the exception in a wider context [41].

A common way of propagating exceptions in Java programs is through exception wrapping (also called chaining): One exception is caught and wrapped in another one which is then thrown instead. Figure 1 shows an exception stack trace which illustrates such exception wrapping. For simplicity, in this paper we will refer to "exception stack trace" as just stack trace. The bottom part of the stack trace is the *root exception* (Figure 1-A), which indicates the first reason (root cause) for the exception thrown (in this case, the computer ran out of memory). The top part of the stack trace indicates the location of the exception manifestation, which we will refer to as the *exception wrapper* in this paper (Figure 1-C). The execution flow between the root exception and the wrapper may include other intermediate exception wrappers (Figure 1-D).

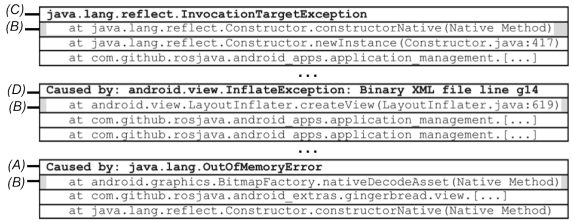

Fig. 1 Example of an exception stack trace in Java.

At all levels, the exception *signaler* is the method that threw the exception, represented in the stack trace as the first method call below the exception declaration (Figure 1-B).

2.3 Best Practices

Several general guidelines have been proposed on how to use Java exceptions [38, 24, 56, 11]. Such guidelines do not advocate any specific exception type, but rather propose ways to effectively use each of them. Based on these, for the purpose of our analysis we compiled the following list of Java exception handling best practices.

I-Checked exceptions should be used to represent recoverable conditions ([38, 24, 56, 11]). The developer should use checked exceptions for conditions from which the caller is expected to recover. By confronting the API user with a checked exception, the API designer is forcing the client to handle the exceptional condition. The client can explicitly ignore the exception (swallowing, or converting it to another type) at the expense of the program's robustness [24].

II-Error represents an unrecoverable condition which should not be handled ([24]). Errors should result from failures detected by the runtime environment which indicate resource deficiencies, invariant failures or other conditions, from which the program cannot possibly recover.

III-A method should throw exceptions that precisely define the exceptional condition ([24, 11]). To do so, developers should either try to reuse the exception types already defined in the Java API or they should create a specific exception. Thus, throwing general types such as a pure java.lang.Exception or a java.lang.RuntimeException is considered bad practice.

IV- All exceptions explicitly thrown by reusable code should be documented. ([38, 24, 56, 11]). For checked exceptions, this is automatically the case. Bloch [11] furthermore recommends to document explicitly thrown run time exceptions, either using a throws declaration in the signature, or using the @throws tag in the Javadoc. Doing so, in particular for public APIs of libraries or frameworks, makes clients aware of all exceptions possibly thrown, enabling them to design the code to deal with them and use the API effectively [45, 56].

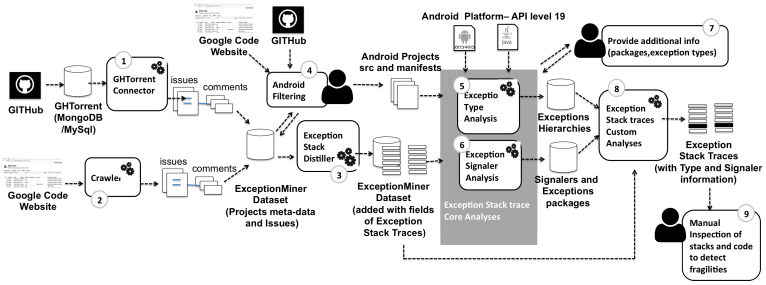

Fig. 2 Study overview.

3 The Repository Mining Study

Our mining study was guided by a general research question: *RQ 1: Can the information available in exception stack traces reveal exception handling* bug hazards *in both the Android applications and framework?* As mentioned before, in this context *bug hazards* are the characteristics of exception-related code that favor the introduction of failures such as uncaught exceptions.

Our study focused on the exception stack traces contained in issues of Android projects (hosted on GitHub and Google Code). To support our investigation, we developed a tool called ExceptionMiner (Section 3.3) which extracts the exception stack traces embedded in issues and combines stack trace information with source code and bytecode analysis. Moreover, we use manual inspection to augment the understanding of stack traces and support further discussions and insights (Section 3.4). In this study we explore the domain quantitatively and highlight interesting cases by exploring cases qualitatively.

Figure 2 gives an overview of our study. First, the issues reported in Android projects hosted on GitHub (1) and Google Code (2) are recovered. Then the stack traces embedded in each issue are extracted and distilled (3). The stack trace information is then combined with source code and bytecode analysis in order to discover the type of the exceptions (5) reported in the stack traces (e.g., error, runtime, checked), and the origin (6) of such exceptions (e.g., the application, a library, the Android platform). Manual inspection steps (4, 7, 9) are used to support the mining process and the search for *bug hazards* (8). The next sections detail each step of this mining process.

Our study focuses on open-source apps, since the information needed to perform our study cannot be retrieved from commercial apps, whose issue report systems and source code are generally not publicly available. Open source Android apps have also been the target of other research [35, 46] addressing *reuse* and API stability.

GitHub		Google Code	
Lable	**% Occurrences**	**Lable**	**% Occurrences**
empty	54.24%	Defect	91.96%
Defect	39.56%	Enhancement	3.16%
Enhancement	0.57%	Task	1.37%
Support	0.52%	empty	1.12%
Problem	0.36%	StackTrace	0.70%
Others	4.74%	Others	1.68%

Table 1 Labels on issues including exception stack traces.

3.1 Android Apps on GitHub

This study uses the dataset provided by the GHTorrent project [25], an offline mirror of the data offered through the GitHub API. To identify Android projects, we performed a case insensitive search for the term "android" in the repositories' names and short descriptions. Up to 23 February 2014, when we queried GHTorrent, this resulted in 2,542 repositories. Running the ExceptionMiner tool on this set we observed that 589 projects had at least one issue containing a stack trace.

Then we performed a further clean up, inspecting the site of every Android project reporting at least one stack trace, to make sure that they represented real mobile apps. During this clean up 107 apps were removed because they were either example projects (i.e., toy projects) or tools to support Android development (e.g. Selendroid, Roboeletric – tools to support the testing of Android apps). The filtered set consisted of 482 apps. This set of 482 projects contained a total of 31,592 issues from which 4,042 exception stack traces were extracted.

Issues on GitHub are different from issues in dedicated bug tracking tools such as Bugzilla and Jira. The most important difference is that there are no predefined fields (e.g. severity and priority). Instead, GitHub uses a more open ended tagging system, where repositories are offered a pre-defined set of labels, but repository owners can modify them at will. Therefore, an issue may have none or an arbitrary set of labels depending on its repository. Table 1 illustrates the occurrences of different labels on the issues including exception stack traces. Regardless of the issue labels, every exception stack trace may contain relevant information concerning the exception structure of the projects analyzed, and therefore can reveal *bug hazards* in the exception-related code. Because of this, we opted for not restricting the analysis to just defect issues.

3.2 Android Apps in Google Code

Google Code contains widely used open-source Android apps (e.g. K9Mail[1]). However, differently from GitHub, Google Code does not provide an API to

[1] K9Mail moved to GitHub but as a way of not loosing the project history it advises their users to report bugs in the Google Code issue tracker: `https://github.com/k9mail/k-9/wiki/LoggingErrors`.

access the information related to hosted projects.[2] To overcome this limitation we needed to implement a Web Crawler (incorporated in the ExceptionMiner tool described next) that navigates the web interface of Google Code projects extracting all issues and issue comments and storing them in a relational database for later analysis. To identify Android projects on Google Code, we performed a similar heuristic: we performed a case insensitive search (on the Google Code search interface) for the term "android". In January 2014, when we queried Google Code, this resulted in a list of 788 projects. This list comprised the seeds sent to our Crawler.

The Crawler retrieved all issues and comments for these projects. From this set, 724 projects defined at least 1 issue. Running the ExceptionMiner tool on this set we observed that 183 projects had at least one issue containing an exception stack trace. Then we performed further clean up (similar to the one described previously) inspecting the site of each project. As a result we could identify 157 Android projects. This set contained 127,456 issues in total, from which 1,963 exception stack traces were extracted. Table 1 illustrates the occurrences of different labels on the issues including exception stack traces. Differently from GitHub, on Google Code most of the issues were labeled as "Defect". However, based on the same assumption described for the GitHub repository we considered all issues reporting stack traces (regardless of their labels).

3.3 The ExceptionMiner Tool

The ExceptionMiner is a tool which can connect to different issue repositories, extract issues, mine exception stack traces from them, distill exception stack trace information, and enable the execution of different analyses by combining exception stack trace information with byte code and source code analysis. The main components of ExceptionMiner are the following:

Repository Connectors. This component enables the connection with issue repositories. In this study two main connectors were created: one which connects to the GHTorrent database, and a Google Code connector which is comprised of a Web Crawler that can traverse the Google Code web interface and extract a project's issues. Project meta-data and the issues associated with each project are stored in a relational database.

Exception Stack Trace Distiller. This component combines a parser (based on regular expressions) and heuristics able to identify and filter exception names and stack traces inline with text. This component distills the information that composes a stack trace. Some of the attributes extracted from the stack trace are the root exception and its signaler, as well as the exception wrappers and their corresponding signalers. This component also distills fine grained information of each attribute such as the classes and packages

[2] Google Code used to provide a Web service to its repositories, but this was deactivated in June 2013 in what Google called a "clean-up action".

associated with them. In contrast to existing issue parsing solutions such as Infozilla, our parser can discover stack traces mixed with log file information[3].

Exception Type Analysis. To support a deeper investigation of the stack traces every exception defined in a stack trace needs to be classified according to its type (e.g. Error, checked Exception or RuntimeException). The module responsible for this analysis uses the Design Wizard framework [12] to inspect the bytecode of the exceptions reported in stack traces. It walks up the type hierarchy of a Java exception until it reaches a base exception type. Hence in this study the bytecode analysis was used to discover the type of each mined exception when the jar file of such an exception was available in the project or in a reused Java library. A specific implementation (based on source code analysis) was needed to discover the exception type when the bytecode was not available. With this module we analyzed all exceptions defined in the Android platform (Version 4.4, API level 19), which includes all basic Java exceptions that can be thrown during the app execution, and exceptions thrown by Android core libraries. Moreover, we also analyzed the exceptions reported in stack traces that were defined by applications and third-party libraries (the tool only analyzed the last version available).

Exception Signaler Analysis. This module is responsible for classifying each signaler according to its origin (i.e., Android Application Framework, Android Libcore, Application, Library). Table 2 presents the heuristics adopted in this classification. To conduct this classification, we provide this module with the information comprising all Java packages that compose: the Android Platform; the Android Libcore; and each analyzed Application. To discover the packages for the first two origins we can use the Android specification. To discover the packages for the third origin, the application itself, this module extracts the manifest files of each Android app, which defines the main packages that the applications consist of. If this file is not available, the tool recursively analyzes the structure of source code directories composing the application, and filters out the cases in which the application also includes the source code of reused libraries. Then, based on this information and using pattern matching between the signaler name and the packages, this module identifies the origin of the exception signalers.

The exceptions are considered to come from libraries if their packages are neither defined within the Android platform, nor in core libraries, nor in the applications. Table 2 summarizes this signaler classification.

3.4 Manual Inspections

In our experiments, the output of the ExceptionMiner tool was manually extended in order to (i) support the identification of packages composing the

[3] In several exception stack traces, the exception frames were preceded by logging information e.g., `03-01 15:55:01.609 (7924): at android.app.ActivityThread.access$600(ActivityThread.java:127)` which could not be detected by existing tools.

Signaler	Description
android	If the exception is thrown in a method defined in the Android platform.
app	If the exception is thrown in a method defined in an Android app.
libcore	If the exception is thrown in one of the core libraries reused by Android (e.g., org.apache.harmony, org.w3c.dom, sun.misc, org.apache.http, org.json, org.xml).
lib	If the exception is thrown in a method that was not defined by any of the elements above.

Table 2 Sources of exceptions in Android

Android platform, libs and apps analyzed in this study (as described previously); and (ii) identify the type of some exceptions reported in issues that were not automatically identified by the ExceptionMiner tool (because they were defined in previous versions of libraries, apps and Android Platform). When the exception could not be found automatically or manually (because they were defined in a previous version of the app or lib), we classified the exception as "Undefined". Only 31 exceptions remained undefined, which occurred in 60 different exception stack traces (see Table 5).

3.5 The Mining Study Results

This section presents the results of our study that was guided by the general research question: *RQ 1: Can the information available in exception stack traces reveal exception handling bug hazards in both the Android applications and framework?* To make the analysis easier, we further refine this question into sub-questions, each one focusing on pieces of information distilled from stack traces, more specifically: (i) the root exceptions (i.e., the exceptions that caused the stack traces); (ii) the exception types (i.e, Checked, Runtime, Error, Throwable) and (iii) the exception wrappings. Hence, this section is centered around the following sub-questions: *RQ 1.1: Can the root exceptions reveal bug hazards?*; *RQ 1.2 Can the exception types reveal bug hazards?*; and *RQ 1.3 Can the exception wrappings reveal bug hazards?*.

In this section each piece of information is analyzed in detail to check whether it can reveal bug hazards in the exception handling code – related to (i) specific violations of the best practices presented in Section 2.3, or (ii) the general use of exception handling to support robust development.

RQ 1.1 Can the root exceptions reveal bug harzards?

After distilling the information available in the exception stack traces, we could find the exceptions commonly reported as the root causes of stack traces. Table 3 presents a list of the top 10 root exceptions found in the study – ranked by the number of distinct projects in which they were reported. This table also shows how many times the signaler of such an exception was a method defined by the Android platform, the Android Libcore, the application itself or a third-party library – following the classification presented in Table 2.

Root Exception	Projects		Occurrences		Android	Libcore	App	Lib
	#	%	#	%				
java.lang.NullPointerException	332	51.96%	1664	27.71%	525	20	836	280
java.lang.IllegalStateException	120	18.78%	278	4.63%	185	31	41	39
java.lang.IllegalArgumentException	142	22.22%	353	5.88%	195	12	95	44
java.lang.RuntimeException	122	19.09%	319	5.31%	203	2	64	51
java.lang.OutOfMemoryError	78	12.21%	237	3.95%	141	16	35	34
java.lang.NoClassDefFoundError	67	10.49%	94	1.57%	10	0	46	37
java.lang.ClassCastException	64	10.02%	130	2.16%	55	0	55	20
java.lang.IndexOutOfBoundsException	62	9.70%	166	2.76%	53	0	93	18
java.lang.NoSuchMethodError	54	8.45%	80	1.33%	10	0	56	14
java.util.ConcurrentModificationException	43	6.73%	65	1.08%	5	0	46	13

Table 3 Root Exceptions occurrences and popularity in analyzed repositories.

We can observe that most of the exceptions in this list are implicitly thrown by the runtime environment due to programming mistakes (e.g., out-of-bounds array index, division-by-zero, access to a null reference) or resource limitations (e.g., OutOfMemoryError). From this set the java.lang.NullPointerException was the most reported root cause (27.71%). If we consider the frequency of NullPointerException across projects, we can observe that 51.96% of all projects reported at least one exception stack trace in which the NullPointerException was the root cause.

The NullPointerException was mainly signaled inside the application code (50%) and the Android platform (31.5%), although we could also find the NullPointerException being signaled by third-party libraries (16.3%). Regarding reusable code (e.g., libraries and frameworks), there is no consensus whether it is a good or a bad practice to explicitly throw a NullPointerException. Some prefer to encapsulate such an exception in an instance of IllegalArgumentException, while others [11] argue that the NullPointerException makes the cause of the problem explicit and hence can be signaled by an API expecting a non-null argument.

The high prevalence of NullPointerException is aligned with the findings of other research [32, 20, 17, 30]. For instance, Kechagia and Spinellis showed that the NullPointerException was the most reported exception in the crash reports sent to BugSense [4] (a bug report management service for Android applications) [30]. Other research on robustness testing [37, 17] shows that most of the automatically detected bugs were due to NullPointerException and exceptions implicitly signaled by the Java environment due to programming mistakes or resource limitations (as the ones found in our study).

Identifying the Concerns Related to Root Exceptions. To get a broader view of the root exceptions of stack traces, we performed a manual inspection in order to identify the underlying concerns related to the most frequently reported root exceptions. Besides the exceptions related to programming mistakes mentioned before, we also looked for exceptions related to concerns that are known as sources of faults in mobile development: concurrency [5] backward compatibility [39], security [19, 55] and resource management (IO, Memory, Battery) [58]. Since it is infeasible to inspect the code responsible for throwing every exception reported in this study, the concern identification for each exception was based on intended meaning of the par-

[4] https://www.bugsense.com/

Concern	% Occurrences on stacks
Programming logic (java.lang and util)	52.0%
Resources (IO, Memory, Battery)	23.9%
Security	4.1%
Concurrency	2.9%
Backward compatibility	5.5%
Specific Exceptions	4.9%
General (Error, Exception, Runtime)	6.7%

Table 4 Identifying the concerns related to root exceptions

ticular exception type, as defined in its Javadoc documentation and in the Java specification. For example: (i) an instance of ArrayOutOfBoundException refers to a programming mistake according to its Javadoc; and (ii) the Java specification lists all exceptions related to backward compatibility [5], such as InstantiationError, VerifyError, and IllegalAccessError.

To perform this concern analysis, we selected a subset of all reported root exceptions, consisting of 100 exceptions reported in 95% of all stack traces analyzed in this study. Hence, based on the inspection of the Javadoc related to each exception and the Java specification, we identified the underlying concern related to each root exception. Table 4 contains the results of this analysis. This table also illustrates the exceptions that could not be directly mapped to one of the aforementioned concerns, either because they were too general (i.e., java.lang.Exception, java.lang.RuntimeException, java.lang.Error) or because they were related to other concerns (e.g., specific to an application or a given library). To ensure the quality of the process, three independent coders classified a randomly selected sample of 25 exception types (from the total 100) using the same list of concerns; the inter-rater agreement was 96%.

This analysis revealed that approximately 75% of the exceptions that caused the stack traces are implicitly thrown by the runtime environment due to mistakes in the programming logic (e.g., out-of-bounds array index, null pointer references) and resource limitations. Although such exceptions do not directly point to violations of the best practices described before (which are related to the explicitly thrown exceptions) they impose a major threat to app robustness, and therefore represent a critical bug hazard to the exception handling code of Android applications. Security and concurrency, which are known to be critical issues for Android apps, raised few of the reported exceptions (less than 5% of the analyzed stack traces).

Exceptions related to programming logic and resource limitations represent a major bug hazard to Android apps – since they represent approximately 75% of the exceptions that caused the stack traces. From this set the NullPointerException is the most prevalent exception.

RQ 1.2 Can the exception types reveal bug hazards?

[5] http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html

Root Type	Android	Libcore	App	Lib	All	%
Runtime	1335	73	1843	690	3894	64.85%
Error	188	46	302	167	691	11.51%
Checked	276	314	313	567	1358	22.61%
Throwable	0	0	2	0	2	0.03%
Undefined	4	0	18	38	60	1.00%
All	1 803	433	2478	1462	6005	

Table 5 Types and origins of root exceptions.

As mentioned before, using the ExceptionMiner tool in combination with manual inspections we could identify the root exception *type* (i.e., RuntimeException, Error, checked Exception) as well as its *origin* – which we identified based on the package names of the signalers in the stack traces (Section 3.3). Table 5 presents the types and origins of root exceptions of all analyzed stack traces.

We can observe that most of the reported exceptions are of type runtime (64.85%); and that the most common origins are methods defined either in the Application (47.3%) or in the Android platform (34.3%). We could also find runtime exceptions thrown by library code (17.7%). We can also see, from Table 5, that in contrast to the other origins, most of the exceptions signaled by Android Libcore (i.e., the set of libraries reused by Android) are checked exceptions. This set comprises: org.apache.harmony, org.w3c.dom, sun.misc, org.apache.http, org.json, org.xml, and javax. Signaling checked exceptions is considered a good practice (see best practice IV in Section 2.3) because by using checked exceptions a library can define a precise *exception interface* [41] to its clients. Since such libraries are widely used in several projects, this finding can be attributed to the libraries' maturity.

Almost 65% of all crashes come from runtime exceptions. Approximately 47% of these originate from the application layer.

Inspecting Exception Interfaces. According to the best practices mentioned before, *explicitly thrown* runtime exceptions should be documented as part of the *exception interface* of the reusable methods of libraries/frameworks. To investigate the conformance to this practice, we first filtered out all the exceptions implicitly signaled by the runtime environment (due to programming mistakes) – since these exceptions should not be documented in the method signature[6]. Then we inspected the code for each method (defined either in the Android Application Framework or in third-party libraries) explicitly signaling a runtime exception. Table 6 presents the results of this inspection. We found 79 methods (both from libraries and the Android platform) that explicitly threw a runtime exception without listing it in the *exception interface*

[6] The filtering was performed in two steps: firstly, we analyzed the bytecode of the JVM and identified all runtime exceptions defined by it (e.g., java.lang.NullPointerExceptoin,java.lang.ArrayIndexOutOfBounds), then if the libraries/framework method was signaling one of such exceptions it was filtered out from the analysis

Origin	stacks	signaler methods	throws clause	@throws
Libraries	205	44	0	1
Android	62	35	0	0
All	267	79	0	1

Table 6 Absence of exception interfaces in methods.

(i.e., using a throws clause in the method signature). From this set only one method (defined in a library) included an @throws tag in its Javadoc – reporting that the given runtime exception could be thrown in some conditions. These methods were responsible for 267 exception stack traces mined in this study.

This result is in line with the results of two other studies [47, 30]. Sacramento et al. [47] observed that the runtime exceptions in .NET programs are most often not documented. Kechagia and Spinellis [30] identified a set of methods in the Android API which do not document their runtime exceptions. One limitation of the latter work is that it did not filter out exceptions that, although runtime, should not be documented because they were implicitly signaled by the JVM due to resource restrictions or violations of semantic Java constraints. When explicitly signaling a runtime exception and not documenting it, the developer imposes a threat to system robustness, especially when such exceptions are thrown by third party code (e.g., libraries or framework utility code) invoked inside the application. In such cases the developer usually does not have access to the source code. Hence in the absence of the exception documentation it is very difficult or even impossible for the client to design the application to deal with "unforeseen" runtime exceptions. As a consequence, the undocumented runtime exception may remain uncaught and lead to system crashes.

Only a small fraction (4%, 267 stack traces) of runtime exceptions are programmatically thrown. Almost none (0.4%, just one) of these were documented. Such undocumented runtime exceptions violate best practices III and IV and reveal a bug hazard to Java/Android apps.

Missing Checked Exceptions in Exception Interfaces. Our exception stack trace analysis revealed an unexpected bug hazard: a checked exception thrown by a native method and not declared in the exception interface of the methods signaling them. The native method in question was defined in the Android platform, which uses Java Native Invocation (JNI) to access native C/C++ code. This exception was thrown by the method getDeclaredMethods defined in java.lang.Class. The Java-side declaration of this method does not have any throws clause, leading programmers and the compiler to think that no checked exceptions can be thrown. However, the C-code implementation did throw a "checked exception" called NoSuchMethodException, violating the declaration. The Java compiler could not detect this violation, because it does not perform static exception checking on native methods. This type of bug is hard to diagnose because the developer usually does not have access

id	**Runtime Exception wrapping an Error**
1	java.lang.RuntimeException - java.lang.OutOfMemoryError
2	java.lang.RuntimeException - java.lang.StackOverflowError
	Checked Exception wrapping an Error
3	java.lang.reflect.InvocationTargetException - java.lang.OutOfMemoryError
4	java.lang.Exception - java.lang.OutOfMemoryError
	Error wrapping a Checked Exception
5	java.lang.NoClassDefFoundError - java.lang.ClassNotFoundException
6	java.lang.AssertionError - javax.crypto.ShortBufferException
	Error wrapping a Runtime Exception
7	java.lang.ExceptionInInitializerError - java.lang.NullPointerException
8	java.lang.ExceptionInInitializerError - java.lang.IllegalArgumentException

Table 7 Examples of Cross-type wrappings

Wrapper	**Root Cause**	**Projects**	**Occurrences**	Android	Java/Libcore	Lib	App
Runtime	Checked	88	148	75	0	38	35
Runtime	Error	46	67	58	0	8	1
Checked	Runtime	17	31	4	0	16	11
Checked	Error	8	9	5	0	1	3
Error	Checked	14	27	6	7	6	8
Error	Runtime	8	17	1	1	1	14

Table 8 Wrappings comprising different exception types.

to the native implementations. Consequently, since it is not expected by the programmer, when such a method throws this exception, the undocumented exception may remain uncaught and cause the app to crash, or may be mistakenly handled by subsumption. The exception stack traces reporting this scenario actually correspond to a real bug of the Android Gingerbread version (which still accounts for 13.6% of devices running Android).

For native methods, even checked exceptions can be thrown without being documented in the exception interface. This violates best practices III and IV and represents a bug hazard hard to diagnose.

RQ 1.3 Can the exception wrappings reveal bug hazards?

Java is the only language that provides a hybrid exception model which offers three kinds of exceptions each one holding an intended exception behavior (i.e., error, runtime and checked). Table 8 presents some wrappings found in this study that include different exception types (i.e., Error, checked Exception and Runtime). Below, we discuss the most important of such "cross-type wrappings" in more detail.

Runtime Exception wrapping a Checked Exception. This wrapping was responsible for 49.5% of the cross-type wrappings. From this set 50% were performed by methods defined by the Android platform. We observe that this is a common implementation practice in the methods of the Android platform. According to Java best practices checked exceptions represent conditions from which the caller is expected to recover. By converting a checked exception in a general Runtime class, besides loosing contextual information about the exception, the client can simply ignore such exception - which can negatively

impact the app robustness. Although it is considered a bad practice, when such wrapping is accompanied by a well defined exception handling policy (which assures that proper handling will be provided within the app) the impact on app robustness can be mitigated.

General catch clauses masking any exception into a general RuntimeException is a common practice in the Android Platform; it violates best practice III and is considered a bug hazard as it loses contextual information about the exception.

Runtime Exception wrapping an Error. From Table 8, we see that most of these wrappings are performed by the Android platform (50.7%). The code snippet below was extracted from Android and shows a general catch clause that converts any instance of Throwable (signaled during the execution of an asynchronous task) into an instance of RuntimeException and re-throws it.

```
try {
   ...
} catch (InterruptedException e) {
   android.util.Log.w(..., e);
} catch (ExecutionException e) {
   throw new RuntimeException("...",e.getCause());
} catch (CancellationException e) {
   ...
} catch (Throwable t) {
  throw new RuntimeException("...", t);
}
```

Table 7 presents examples of exceptions that were actually wrapped in this code snippet: java.lang.RuntimeException wrapped an java.lang.OutOfMemoryError and a java.lang.StackOverflowError. Such Errors result from failures detected by the runtime environment which indicate resource deficiencies from which the program cannot possibly recover. Hence, "hiding" an unrecoverable condition into a runtime exception can lead to a scenario where the developer tries to handle such an exception, leading the program to an unpredictable state. This kind of wrapping should be avoided, since it represents a serious bug hazard.

Checked Exception wrapping an Error. Most of these wrappings were also caused by the reflection library used by applications' methods. The methods responsible for the wrappings were also native methods written in C. Table 7 illustrates some of these wrappings — some of them are masking an OutOfMemoryError into a checked exception. On the one hand, by confronting the API user with a checked exception, the API designer is forcing the client to handle the exceptional condition. On the other hand, according to the Java specification Errors are not supposed to be caught. In this case, even if the exception is caught by a handler, the problem that trigged the Error remains:

there is not enough memory to execute the app. This kind of wrapping is even more dangerous than the previous one and may lead to the "exception confusion" described next.

Error wrapping Runtime and Checked Exceptions. Table 7 illustrates examples of instances of Error wrapping instances of RuntimeException. Although such a wrapping mixes different exception types, since there is no obligation associated with handling runtime exceptions, it does not violate the aforementioned best practices.

On the other hand, the inspection also revealed instances of Error wrapping checked exceptions. Such wrappings were mostly performed by Java static initializers. If any exception is thrown in the context of a static initializer (i.e., static block) it is converted into an ExceptionInitializerError at the point where the class is first used. Table 7 also illustrates examples of such wrappings. Although such a wrapping may represent a design decision, it violates the best practices related to checked exceptions and errors as it mixes the intended handling behavior associated with both types.

We can also observe that some stack traces include successive cross-type wrappings, such as: Runtime - Checked - Runtime - Checked - Runtime - Checked - Runtime. Hence, although some of these wrappings may be a result of design decisions, the mis-use of exception wrappings may make the exception handling code more complex (e.g., the multiple wrappings) and error-prone, and lead to "exception confusion". To illustrate this problem we can use one of the wrappings discussed above. When the developer is confronted with a checked exception, the designer of the API is telling him/her to handle the exceptional condition (according to the Java Specification and best practices). However, such an exception may be wrapping an Error such as an OutOfMemoryError, which indicates a resource deficiency that the program cannot possibly recover from. Hence, trying to handle such an exception may lead the program to an unpredictable state.

Cross-type exception wrappings are common. They represent a bug hazard once they violate the semantics of Java's original exception design, detailed in best practices I and II (e.g., when mapping unrecoverable Errors to other types of exceptions).

4 The Developers' Perspective

As previously mentioned, the goal of this work is to obtain a thorough understanding of common exception handling *bug hazards* that app developers face. In the first phase of this work, we conducted a mining study which identified common exception handling bug hazards in app development. In the second phase of this work, we set up an exploratory qualitative investigation and surveyed Android developers on how they perceive the bug hazards detected in the mining study. The scope of our study is GitHub—using our GHTorrent

database [25], we aimed our survey at developers from Android projects available in GitHub for which issues were mined during the first phase of this study. Next sections detail this exploratory qualitative investigation.

4.1 Research Questions

The overall research question guiding our exploratory survey is the following: *RQ 2: How do developers deal with the exception handling code in Android apps and what are the developers' perspectives about the main bug hazards found during the mining study?* This general research question has been broken into a set of research questions that are answered by the exploratory survey.

When developing Java-based applications it is inevitable to deal with exceptions. Hence, our first question explores how developers deal with the exception handling code in Android development:

RQ 2.1: *How do developers deal with exception handling code while developing Android apps?*

To make the analysis easier, we further refine this question into subquestions, as follows. To investigate whether the development of exception handling code is a daily concern for developers or it is something that developers rarely face and for that reason do not care much, the following subquestions are added: *How often do developers handle exceptions? How often do developers throw exceptions?* And, to evaluate developers knowledge concerning EH best practices, we also investigated: *Do developers know about Java EH best practices and/or Android specific EH best practices?*

The subsequent research questions focus on the developers' perspectives about the main bug hazards found during the mining study. We address these questions by presenting a set of code snippets in which the bug hazards are present, asking developers what they would do when faced with such scenarios. The questions related to bug hazards are as follows:

RQ 2.2: *How do NullPointerExceptions impact the development of robust Android apps?*

RQ 2.3: *How do cross-type wrappings impact the development of robust Android apps?*

RQ 2.4: *Are developers aware of the robustness threats caused by JNI undocumented checked exceptions?*

Our last research question addresses whether the exception handling code helps with the development of robust Android applications, and what developers usually do to prevent apps from crashing. The motivation behind this question was to discover common practices for dealing with uncaught exceptions and to assess developers perspective about the role of the exception handling mechanism in the development of robust apps. Hence, the last question is as follows:

RQ 2.5: *How does the exception handling code affect the development of robust apps and how do developers prevent apps from crashing?*

4.2 Protocol

Given the exploratory nature of our research, we used methods from Grounded Theory [14] to answer some of our research questions. Since our aim is to learn from a large number of developers, we use surveys which are known to scale well. The survey is split into two logical sections; the motivations behind each logical section are: (i) questions aiming at learning from developers about the usage of the exception handling code in app development; and (ii) questions focusing on getting developers' perceptions about the bug hazards detected in the first phase of this study.

The survey comprises multiple choice or Likert-scale questions and open-ended questions. The multiple choice questions are intermixed with open-ended questions to further elicit the developer's opinions. Moreover, the survey also contains Likert-scale questions to force participants to make a choice. Overall, the survey includes 13 open-ended questions, 5 Likert-scale questions and 10 multiple choice questions. The respondents could complete the survey in about 15 minutes.

We used grounded theory coding to iterate through the open-ended survey responses. The grounded theory coding used in this study consists of two phases: (1) initial coding entails a close reading of the data and (2) later we used focused coding [14] to pinpoint and develop the most salient themes [14] in the analyzed data. Answers to different questions in the survey were coded separately. For all open-ended questions in the survey, coding was done by two researchers until saturation was reached. Disagreements were discussed and resolved as part of the coding, thus, we are unable to report level of agreement. In all cases, the first author was one of the two coders, and one of the other authors coded the data as well in close collaboration with the first author until saturation was reached. The first author then went through the remaining responses to assign codes to responses.

4.3 Participants

In the first phase of this study, described previously, we mined exception stack traces available in issues reported for several Android projects hosted on GitHub and Google Code. To ensure that our sample consists of repositories that were real Android apps, we had to inspect every project site and discard toy-programs and non-Android repositories. For each repository, we extract the developers whose emails are registered. We emailed the 1,824 developers and received 71 valid answers. The response rate was 3.9% – although the response rate achieved in our study is low, it is in line with similar surveys reported in the literature (e.g., [34, 29, 7, 33, 27]) whose response rates mostly vary between 2% and 4%. The majority of our respondents have more than 2 years of Java development experience (87.3%) and of Android development (85.9%) – see Figure 3.

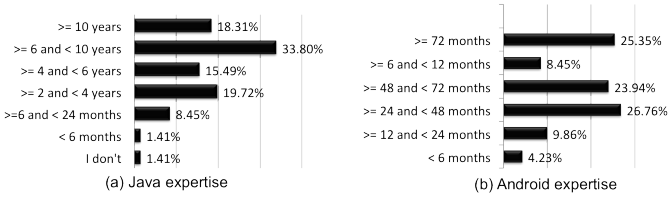

Fig. 3 Developers expertise in Java and Android development.

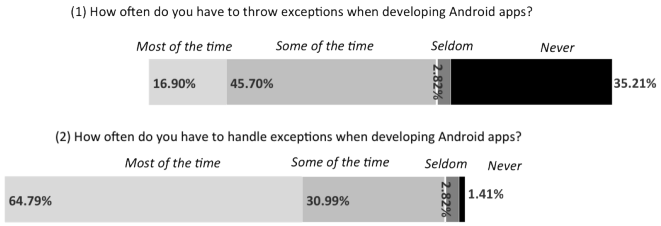

Fig. 4 How often developers throw and handle exceptions during app development.

4.4 Findings

In this section, we present our findings per research question. To illustrate the different aspects of each finding, we provide a selection of quotes from the exploratory survey. To enable traceability, each respondent has an identification which can be traced in our database using the following convention: D#. For instance, D1 corresponds to an answer provided by developer 1.

RQ 2.1: How do developers deal with the exception handling in Android development?

This first research question explores the ways in which developers deal with the exception handling code, either throwing or handling exceptions, while developing an Android app. Developers were asked about the frequency at which they develop exception handling code and whether or not they adopt best practices while developing.

Developers were first asked how often they dealt with the exception handling code – see Figure 4. Many of our survey respondents recognized they have to handle exceptions most of the time during Android app development (64.8%). However, the frequency at which they throw exceptions inside apps is smaller; most of the survey respondents said they throw exceptions only some of the time or seldom (80.3%). It shows that although we can try to avoid throwing exceptions, it is almost always inevitable to handle the exceptions thrown by the Android platform and reused libraries.

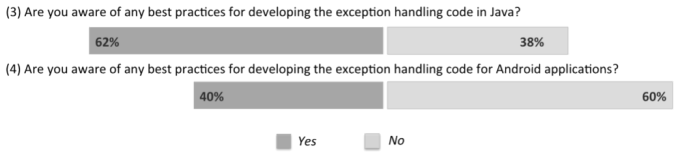

Fig. 5 Summary of the survey questions related to EH best practices.

As mentioned before several general guidelines have been proposed on how to use Java exceptions. Section 2.3 compiled a list of some exception handling (EH) best practices that guided our mining study. In this survey the developers were also asked if they knew about and used Java EH best practices – see Figure 5. Most of them said that they knew and adopted Java EH practices (68%). From this set the most mentioned best practices are presented in Table 9 – considering that one developer mentioned more than one practice. Most of the best practices mentioned were the ones cited in Section 2.3. For instance: *"Don't throw [pure] RuntimeException (in most cases), don't catch OutOfMemoryError, etc." [D42].* Some of the practices mentioned, however, are not well known Java EH best practices but seem to be used by some Android developers to better deal with exceptions in the Android app context, such as the use of a crash fast approach[7] – *"Crash is good for hinting developer. Fast crash fast solve." [D51].* According to this approach, developers should let the application crash as soon as an unexpected situation happens. The motivation behind this approach is that the developer will receive the crash information and fix the potential bug that caused it. On the other hand, if the developer lets the app continue to run after an exception had happened, the application may enter an inconsistent state. In such scenario, the time between the fault being exercised and the failure (the fault manifestation) increases, which may impair finding and fixing the potential bug that caused the failure. Moreover, some respondents favor checked exceptions instead of runtime to represent the exceptional conditions in Android apps – *"RuntimeExceptions are not particularly well suited for Android's robustness, as any uncaught one would cause a crash." [D26]*; *"Checked exception help, unchecked exceptions don't." [D25]*; *"avoid unchecked exceptions except if the case is really abnormal and should not happen [...]"[D45].*

The developers who informed us about their EH best practices were also asked whether they knew about any EH best practices specific to Android – see Figure 5. Most of them (43%) mentioned that they applied the same best practices that they used in Java programs. Only some of them mentioned that they adopted specific best practices such as: (i) the use of crash report tools (21%) – to notify the developer about the uncaught exceptions that happen

[7] `http://www.slideshare.net/pyricau/crash-fast`

Top Java EH Best Practices	#	%
Use specific handlers / don't catch generic exceptions	9	23%
Don't swallow Exceptions	7	18%
Don't throw Runtime / Favor Checked exceptions	4	10%
Do not use exception for normal flow control	4	10%
Free Resources in finally-blocks	4	10%
crash fast	3	8%
crash report tools	2	5%
Don't catch Errors	2	5%

Table 9 Top mentioned Java EH best practices – 40 non-empty responses.

Top Android EH Best Practices	#	%
Same as Java	6	43%
Crash report tools	3	21%
Add a global exception handler (UncaughtExceptionHandler)	2	14%
Use checked exceptions	1	7%
Use appropriate exception messages	1	7%

Table 10 Top mentioned Android-specific EH best practices – 14 non-empty responses.

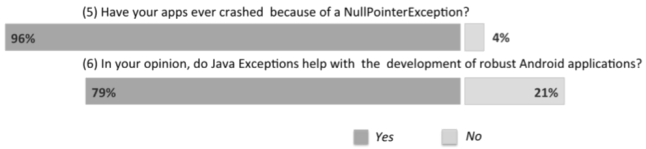

Fig. 6 Summary of the survey questions relates to the null pointer problem (RQ 2.2) and the effect of EH on app robustness (RQ 2.5).

in the application code; and (ii) the use of a global exception handler (14%) – the UncaughtExceptionHandler advised in the Android documentation[8].

Most of the respondents (64.8%) recognize that they handle exceptions most of the time during the app development. Most of them (62%) mentioned that they adopt EH best practices.

RQ 2.2: How do the NullPointerExceptions impact the development of robust Android apps?

Developers were asked if their apps ever crashed because of a NullPointerException – see Figure 6. The vast majority of respondents (96%) said that their apps crashed at least once due to a NullPointerException. Their answers are aligned with the high prevalence of NullPointerExceptions found in the mining study – in Table 3 in Section 3.5 we can see that NullPointerExceptions were the main cause of the mined exception stack traces.

Developers reported some reasons for the high prevalence of crashes caused by NullPointerExceptions. The activity and fragment lifecycle was one of them – *"Android destroys and recreates itself all of the time (especially during screen rotation) and if you do not handle that it will crash on you every time. with*

[8] `http://developer.android.com/reference/java/lang/Thread.UncaughtExceptionHandler.html`

Top Reasons	#	%
activity/fragment life cycle	9	47%
framework complexity	6	32%
API / Backward compatibility	2	11%
memory issues	1	5%
no layers to catch runtimeexceptions	1	5%

Table 11 Top reasons why NullPointerExceptions are more frequent in Android apps according to 19 respondents.

the complexity of an activity with a fragment that has fragments and each of those fragments has custom objects and variables that need to be either retained (so saved and put back) or recreated such as views it can get complex if you don't have an understanding of how the android life cycle works." [D43]. Another developer also mentioned that: *"They [NullPointerExceptions] can happen pretty much anywhere. The Android Fragment system comes to mind in this case. Often, it is possible to find yourself in a state where getActivity() is null within the Fragment during certain points in the life cycle, and that is something I have to plan for. This might have been avoidable under a different structure."[D56].*

The framework complexity was also mentioned as a major cause of null pointer crashes – since several methods of the Android framework return null, and there is not enough documentation to inform the developer when using such methods. Some respondents mentioned: *"The Android framework is what adds the complexity in figuring out what caused an exception because more often than not, the error is triggered from the framework as a result of something else you did." [D58]*; *"Java is simple language, and the problem arises when you are using third party frameworks build on Java (or using Android which provide lots of new classes). And the problem is you don't always know if specific method can return null object or only valid values."[D7].* Moreover, a few respondents also answered that such NullPointerException crashes can be caused by backward compatibility as well as memory issues as illustrated in Table 11.

Almost half of the developers mentioned, however, that NullPointerExceptions are common in Android development since they are common in Java standard development as well. One responded even mentioned: *"Honestly, there are a lot of very unskilled Java programmers out there writing Android apps. When they encounter NPE, they tend to null-check that variable, which just puts a bandage on the problem and causes other failures (usually also NPE's) later on in the application's lifecycle. This is nothing specific to Android, it's just how Java works."[D42]*

Developers were also asked about what they usually do to prevent NullPointerExceptions from happening when their app is in production. Table 12 presents the ways for preventing NullPointerException in Android apps most cited by developers. Most of the developers mentioned including null-checks in the application code (59%), mostly when checking the return of a method and as method guard conditions. Many respondents (39%) also answered that instead of just sprinkling the code with null-checks when a NullPointerExcep-

Top Ways of Preventing NullPointerExceptions	#	%
null-checks	36	59%
investigate/fix the cause	24	39%
@Nullable @NotNull	8	13%
catch null-pointer (mistake)	3	5%
initialize/use default variable	3	5%
new control-flow for null	3	5%
avoid using nulls / avoid to use methods can throw null	2	3%
static analysis	2	3%
automated testing	1	2%

Table 12 Top ways for preventing NullPointerException in Android apps – 61 non-empty responses.

tion happens they investigate the cause of the null reference and work to fix it, instead of just adding a null check. *Determine why the object was null and attempt to fix this situation. In the case of external API calls which return null, then check for null (the quick-and-dirty way). For internal calls, use @NotNull and @Nullable annotations to provided more guidance on when an object "may be" and "should never be" null." [D42]*

Some respondents also mentioned using annotations such as @Nullable and @NotNull, provided by Android Studio 0.5.5; such annotations guide the developer about what can and cannot be null during the application lifecycle. Although catching the NullPointerExceptions is considered a bad practice some respondents said that they do it to prevent crashes caused by NullPointerExceptions. Such behavior was mentioned by one of the respondents as the behavior adopted by non-experienced developers to prevent such crashes – which is undoubtedly a mistake. Some respondents also suggested to replace the NullPointerException by IllegalArgumentException when a method should not receive a null as parameter. *"Probably the most common exception, as such the reasons and fixes are super varying. But in general, null checks should be utilized and illegal state or argument exceptions should be used with an appropriate message. Which will communicate useful information without the confusion an NPE would normally cause."[D48].* This is a good exception handling practice mentioned by Bloch [11]. Only a few developers mentioned coding standards and static code analyzers to prevent it (2%).

Almost 96% of the developers recognized that their apps have already crashed due to a NullPointerException. They mentioned that the Android application life-cycle (47%) and the framework complexity (32%) may favor such uncaught NullPointerExceptions to happen in the Android context.

RQ 2.3: How do Cross-Type Wrappings impact the development of robust Android apps?

Some of the best practices are related to the different ways an exception should be handled according to whether it is a checked or an unchecked exception. When the designer of an API specifies that a method throws a checked

exception, it is telling the caller of the API that such an exception should be handled.

To assess the developers' perspective on whether checked and unchecked exceptions should be handled differently inside the app, the developers were presented with two pieces of code: one in which a RuntimeException was caught and another one where an IOException was caught; both in the context of Activity life-cycle methods. Then the developers were asked whether the way to handle a runtime exception should be different from the way to handle a checked exception.

This question was motivated by the fact that during the mining study we could identify that many checked exceptions were just wrapped in a runtime exception and re-thrown, as a way to bypass any kind of handling. When asked about how to deal with a checked exception signaled by a method (invoked in the context of the onPause() activity method), most respondents (63%) said that they should add a try-catch block surrounding the method invocation - see Table 13. One of the respondents mentioned that *"If a method signature declares that it throws an exception, it means that the caller should handle the exception."[D29]*; *"because it is checked exception, you should expect it to happen rather than just crash"[D70]*. However, most of the respondents also said that to deal with a method that is signaling a runtime exception we should surround such a method with a try-catch block (59%) - see Table 14.

However, what developers mentioned as handling actions (what is performed inside the catch clause to deal with the exception) differs in both cases. For runtime exceptions most of the handling actions were dedicated to present more specific error messages and prevent the app from abruptly crashing. *"[...] They signal exceptional behavior that may not be recoverable, so they offer a useful way to log and gracefully crash" [D19]. "Giving the user meaningful feed back is important, so the more specific you can be about what went wrong, the better."[D66]*. Some developers also mentioned to use a toast to support this task[9]. Few developers also mentioned that the runtime exception should flow upstream to the framework so it could crash the app.

On the other hand, some handling actions mentioned for checked exceptions were: (i) retrying the same operation; (ii) involving the user in finding a solution to the exception condition such as opening a pop-up and asking the user to define a new place to save the file (the example presented to them involved a IOException being signaled); and also (iii) presenting an error message to the user.

We can observe that the handling actions associated with checked exceptions focused more on crash prevention than the ones related to runtime exceptions. *"Its very important in android that we gracefully handle an exception. Try not to crash an application."[41]*. It was also interesting to observe that the developers suggested involving the user and solving the exceptional condition represented by a checked exception – *"We should try alternative saving*

[9] A toast is an Android component that provides simple feedback about an operation in a small popup – `http://developer.android.com/guide/topics/ui/notifiers/toasts.html`

Top Ways of Handling Checked Exceptions	#	%
Add a try-catch block	25	63%
Present error message	7	18%
Involve the user in solving	5	13%
Retry	5	13%
Investigate the cause	3	8%
Same way as runtime	3	8%
Add a throws declaration / pass upstream	2	5%

Table 13 Top ways of dealing with a checked exception signaled in the context of an Activity method – 40 non-empty responses.

Top Ways of Handling Runtime Exceptions	#	%
Add a try-catch block	37	59%
Present error message	14	22%
Log the exception	12	19%
Throw a checked exception	12	19%
Let it crash / crash fast	6	10%
Swallow the exception	5	8%
Add a throws declaration	5	8%
Report crash	3	5%
Use Toast	3	5%

Table 14 Top ways of handling runtime exceptions – 63 non-empty responses.

ways and if it fails we should prompt the user for a new location and always keep the user informed"[D11]. "You still need to catch an IOException to prevent a crash. However, because it is related to saving a file, you may need to reset any data within that catch statement and either alert the user that it has failed or perform a limited retry (in the case of an HTTP upload or something prone to server-side failure)."[D5].

Although there is a long lasting debate about the pros and cons of checked and unchecked exceptions, the survey revealed that many Android developers considered checked exceptions as a way of using exceptions that can prevent uncaught exception crashes – since in order for a checked exception to flow upstream and crash the app the developers need to explicitly do so.

After presenting two different code snippets in which a checked and a runtime exception were thrown, developers were presented with a code snippet in which a cross-type wrapping was performed[10] – as illustrated below. Then developers were asked whether such cross-type wrapping could affect the application robustness in some way.

```
@Override
public void onPause() {
   try {
      ...
    } catch (Exception e) {
         throw new RuntimeException("...", e);
    }
}
```

Table 15 illustrates the most mentioned reasons of why cross-type wrapping may affect app robustness; 18% of the respondents mentioned that the app

[10] This cross-type wrapping was found in several applications during the mining study as well as in some classes of the Android framework.

Top Reasons Why Cross-Type Wrapping Affect Robustness	#	%
impairs proper handling (loses exception information)	12	24%
uncaught will crash the app	12	24%
app will crash anyway	9	18%
should catch / handle properly (do local recovery)	5	10%
treat all exceptions as critical	5	10%
useless rethrow	2	4%
activity methods cannot throw exceptions	1	2%

Table 15 Top reasons why cross-type wrapping may affect app robustness – 49 non-empty responses.

would crash anyway, regardless of whether the exception was wrapped or not. Most of the respondents, however, mentioned disadvantages of such wrapping, such as: (i) it impairs proper handling since wrapping in a general exception loses information about the exception situation to be handled (24%); such wrapping treats all exceptions as critical – in other words, every exception will remain uncaught and will lead to an app crash – and it does not allow the developer to perform a proper handling (e.g. retry) for exception types that are not critical. A respondent emphasized that: *"this treats all exceptions as critical failure, even though they might not all be unrecoverable"[D70].* Another developer alerted that: *"Rethrowing a checked exception as an unchecked exception is the worst choice; it throws away any value of checked exceptions, elevates the exception out of almost all handling, and does absolutely nothing helpful in the process. Further, catching 'Exception' suggests that the author doesn't know what exceptions might occur and wrongfully assumed that this solution is somehow safer than doing nothing."[D19]*

Wrapping a checked exception in a runtime exception was considered a bad practice by many respondents; since it loses exception information it impairs proper handling (24%), and since the runtime exception may remain uncaught it may crash the app (24%).

RQ 2.4: Are developers aware of the robustness threats caused by JNI undocumented checked exceptions?

The respondents were presented with a piece of code in which a non-declared checked exception has appeared, and they were asked if they knew any reason for this to happen. Only 3 respondents out of 71 (4% of respondents) were aware of the fact that there are ways a method can throw a non-declared checked exception, such as JNI/native code, reflection, and directly changing the bytecode/dalvik code, which bypass the compile-time checking. One of them said: *"I am guessing this is because the exception is thrown from native code (i.e., C++ code in the JVM) where Java correct-ness semantics rules do not always apply." [D42].*

Most of the respondents, however, were not aware that a method could throw a checked exception without declaring it in the method's signature. Most respondents mentioned that if a method throws a non-declared exception it

Top Reasons	#	%
For Positively Affecting Robustness		
Improves error diagnosis	10	15%
Anticipated erroneous situation help writing more robust code	9	13%
Exceptions only for unrecoverable behavior	7	10%
Checked exceptions force handling	6	9%
Useful to gracefully crash	4	6%
For Negatively Affecting Robustness		
Crashes the app if not handled	7	10%
Makes debugging harder	2	3%
Unchecked/runtime exceptions may crash the app	2	3%
NullPointerExceptions can happen anywhere/are tricky to avoid	2	3%
VM inefficiency	1	1%

Table 16 Top 5 reasons why exception handling helps or impairs the development of robust apps

must be a RuntimeException. We could observe that this exception handling bug hazard detected during the mining study, although representing a threat to app robustness, is hardly known by the Android developers involved in this survey.

Only 4% of the developers were aware of the robustness threats caused by JNI undocumented checked exceptions.

RQ 2.5: How does the exception handling code affect the development of robust apps and how do developers prevent apps from crashing?

Developers were also asked whether the exception handling code helped with the development of robust Android applications – see Figure 6. Although most of the developers answered positively (78.9%), when asked to provide one or more reasons for their answers, most of the respondents also provided reasons why the exception handling code can sometimes impair the robustness.

Since positive and negative reasons were provided intermixed we analyzed both sets of answers and ranked the top reasons why exception handling can negatively or positively affect robustness according to developers' perceptions. Table 16 presents the top reasons why the exception handling code may or may not help the development of robust apps.

Although most of the respondents answered that exception handling code helped with the development of robust Android applications, almost every respondent pointed out a drawback associated with the exception handling code, saying that if it is not used with care the exception handling code may lead to app crashes. *"Any uncatched exception will crash the app. There are many unknown exception thrown by the system, that are only happen once in a lifetime like IllegalStateException: eglMakeCurrent failed EGL_BAD_CONTEXT"[D32]. "The lifecycle of activitiesfragments etc make it harder to work out what order things are called in and so exceptions can occur because you didn't realise that another method isn't called."[D70]*

Developers were also asked to rank the main causes of crashes. Approximately 68% of the respondents ranked the exceptions signaled by programming

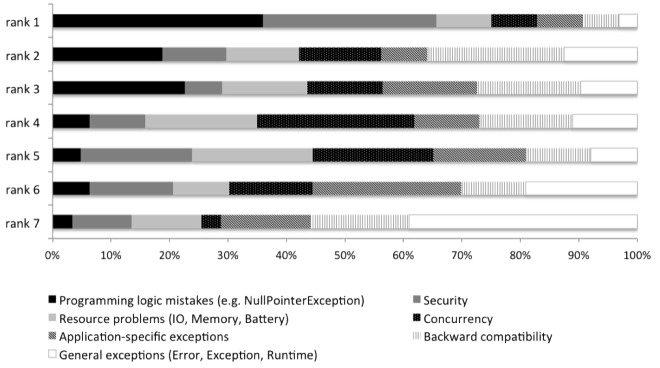

Fig. 7 Developers' perspectives on the main causes of crashes.

logic mistakes, e.g. NullPointerExceptions, as the Top 1 and Top 2 cause of crashes, as illustrated in Figure 7. In our mining study the exceptions signaled due to programming logic mistakes (java.lang and util) were the causes of 52% of the exception stack traces found. Hence, the developers' perspective is aligned with what was revealed in the mining study.

Finally developers were asked how they prevent their apps from crashing. Most of the respondents answered that they prevent apps from crashing by handling exceptions/errors (27%). They mentioned strategies such as: handling exceptions in all entry points/all over; using catch-all clauses; and adding try-catch blocks around risky methods. Different from conventional Java programs, an Android app is composed of several entry points, each window (i.e. Activity) is a potential entry point in which exceptions can arise leading to an app crash. This explains the importance of handling exceptions in all entry points, and using general catch clauses which sometimes cannot prevent crashes but allow the developer to present a detailed error message to the user before crashing. *"You can wrap the main function in java in try/catch in android you can't..."[D12]. "Though not particular to Android, most Android apps have several different entry points: Activities, Services, etc, can be started by different services; same thing with events. It may be hard to ensure that all this cases are properly handled." [D46]*

Many respondents also said that they used testing approaches to prevent crashes – few of them mentioned crowd testing (3%), an emerging approach in testing apps. Moreover, null checks were also mentioned by 15% of the developers as a way to prevent crashes. The adoption of coding styles was also pointed out by 14% of respondents as a way to prevent crashes. Finally, crash report tools (e.g. ACRA [11]) were reported as a way to prevent crashes. Once such tools notify developers about current crashes, the application can be

[11] code.google.com/p/acra/

Top Ways of Crash Prevention	#	%
handling exception/error	19	27%
testing	14	20%
null check(s)	11	15%
- null check annotations (@Null and @NotNull)	3	4%
coding style(s)	10	14%
crash report / crash report tools	8	11%

Table 17 Top ways of preventing crashes – 71 non-empty responses.

fixed and hence future crashes (caused by what was fixed) could be prevented. Table 17 presents the themes for the strategies for crash prevention mentioned by developers.

Developers recognized that exception handling affects the application robustness in two different ways. On the one hand, anticipated erroneous situations help writing more robust code. On the other hand, any uncaught exception will lead to an app crash.

5 Discussion

"Everybody hates thinking about exceptions, because they are not supposed to happen" (Brian Foote)[12]

This section discusses the lessons learned from the mining and the survey studies. The mining study revealed a set of bug hazards – such as (i) the cross-type wrappings; (ii) the abundance of null pointer problems; and (iii) the undocumented runtime exceptions signaled by third-party code, which were confirmed by developers during the survey. Here, we discuss the identified bug hazards, we present how the developers perceived them and point out at what developers can do in order to deal with them.

5.1 The exception handling confusion problem

When (mis)applied, exception wrapping can make the exception-related code more complex and lead to what we call the *exception handling confusion problem*. This problem can lead a program to an unpredictable state in the presence of exceptions, as illustrated by the scenario in which a checked exception wraps an OutOfMemoryError. Currently there is no way of enforcing Java exception type conventions during program development. Hence, further investigation is needed on finding ways to help developers in dealing with this problem, either preventing odd wrappings or enabling the developer to better deal with them.

[12] Brian Foote shared his opinion in a conversation with James Noble – quoted in the paper: hillside.net/plop/2008/papers/ACMVersions/coelho.pdf

Furthermore, only some of the Android developers surveyed in this study were aware of the robustness threats caused by cross-type wrappings as the one cited above. This calls for empirical studies on the actual usefulness of Java's hybrid exception model.

5.2 On the null pointer problem.

The null reference was first introduced by Tony Hoare in ALGOL W, which after some years he called his "one-billion-dollar mistake" [4]. In this study, the null references were, in fact, responsible for several reported issues – providing further evidence to Hoare's statement. Moreover many of the survey respondents recognized that NullPointerExceptions were one of the main causes of application crashes since they can happen almost anywhere in the application code, and to make things worse the life-cycle of Android apps in which objects are constantly recreated (i.e., Activity and Fragment classes) favors this kind of exception to happen added to the fact that many Android framework methods return null without making this return type explicit in the documentation. Such observations emphasizes the need for solutions to avoid NullPointerExceptions, such as: (i) lightweight intra-method null pointer analysis as supported by Java 8 @Nullable annotations[13]; (ii) inter-method null pointer analysis tools such as the one proposed by Nanda and Sinha [42]; or (iii) language designs which avoid null pointers, such as Monads [53] (as used in functional languages for values that may not be available or computations that may fail), to improve the robustness of Java programs. Some of the Android developers mentioned that @Nullable annotations could be helpful to deal with NullPointerExceptions and consequently prevent app crashes caused by them.

5.3 Preventing uncaught exceptions

In this study we could observe undocumented runtime exceptions thrown by third party code, and even undocumented checked exceptions thrown by a JNI interface. Such undocumented exceptions make it difficult, and most of the times infeasible, for the client code to protect against "unforeseen" situations that may happen while calling library code. In the survey-based study Android developers were asked about ways to prevent application crashes which are mainly caused by uncaught exceptions. Many of them emphasized the importance of handling exceptions to prevent crashes but also mentioned how difficult it is to handle every specific exception that can happen. Several developers follow a more reactive behavior against crashes: they advocate the use of crash report tools, and once the crash happens for the first time, it is reported

[13] Already supported by tools such as Eclipse, IntelliJ, Android Studio 0.5.5 (release Apr. 2014) to detect potential null pointer dereferences at compile time.

and the application can be changed to better cope with the exceptions that caused the crashes, consequently preventing future similar crashes.

One may think that the solution for the uncaught exceptions may be to define a general handler, which is responsible for handling any exception that is not adequately handled inside the applications. Although this solution may prevent the system from abruptly crashing, as mentioned by some of the surveyed developers such a general handler will not have enough contextual information to adequately handle the exceptions, beyond storing a message in a log file and restarting the application. Such a handler cannot replace a carefully designed exception handling policy [45], which requires third-party documentation on the exceptions that may the thrown by APIs used. Since documenting runtime exceptions is a tedious and error prone task, this calls for tool support to automate the extraction of runtime exceptions from library code. Initial steps in this direction have been proposed by van Doorn and Steegmans [52].

5.4 Mining Study – Threats to Validity

Internal Validity. We used a heuristics-based parser to mine exceptions from issues. Our parsing strategy was conservative by default; for example, we only considered exception names using a fully qualified class name as valid exception identifiers, while, in many cases, developers use the exception name in the issue description. Conservative parsing may minimize false positives, which was our initial target, but also tends to increase false negatives, which means that some cases may have not been identified as exceptions or stack traces. Our limited manual inspection did not reveal such cases. Moreover, in this study we manually mapped the concerns related to exceptions. To ensure the quality of the analysis, we calculated the interrater agreement after three independent raters classified a randomly selected sample (of 25 exception types from the total of 100); the interrater agreement was high (96%).

The process for identifying the type of exceptions reported in stack traces may not be completely accurate. Specifically, we performed the selection of the version of the exception source code or bytecode to analyze, as follows: (i) for all exceptions signaled by the Android Platform and the Java Environment the analysis was based on Version 4.4 of Android platform (API level 19); (ii) for the exceptions signaled by applications, the analysis considered the last version of the application source-code available on Github/Google code; and (iii) for all exceptions signaled by third-party libraries the type analysis considered the latest bytecode version available on the app repository. When the exception could not be found automatically or manually (based on the chosen version), we classified the exception as "Undefined". Only 31 exceptions remained undefined, which occurred in 60 different exception stack traces out of 6,005 mined stack traces. Hence, the exception type could not be accurately identified in 1% of the mined exception flows. There should be situations where the exception type (i.e., checked or runtime) changes from one version to an-

other. Although we believe that it will not happen very often, we did not investigate this issue.

External Validity. Our work uses the GHTorrent dataset, which although comprehensive and extensive is not an exact replica of GitHub. However, the result of this study does not depend on the analysis of a complete GitHub dataset. Instead, the goal of our study was to pinpoint *bug hazards* in the exception-related code based on exception stack trace mining of a subset of projects. We limited our analysis to a subset of existing open-source Android projects. We are aware that the exception stack traces reported for commercial apps can be different from the ones found in this study, and that this subset is a small percentage of existing apps. Such threats are similar to the ones of other empirical studies which also use free or open-source Android apps [35, 39, 46]. Moreover, several exception stack traces that support the findings of this study referred to exceptions coming from methods defined in the Android Application Framework and third-party libraries. Additionally, the *bug hazards* observed in this study are due to characteristics of the Java exception model, which can impose challenges to the robustness of not only Android apps but also to other systems based on the same exception model.

Another threat relates to the fact that parts of our analysis are based on the availability of stack traces in issues reported on GitHub and Google Code projects. In using these datasets, we make an underlying assumption: the stack traces reported in issues are representative of valid crash information of the applications. One way to mitigate this threat would be to access the full set of crash data per application. Although some services exist to collect crash data from mobile applications (e.g., ACRA [14], Google analytics [15], Bugsense, Bugsnag [16]), they do not provide open access to the crash reports of their client applications. In our study, we mitigated this threat by manually inspecting the source code associated with a subset of the reported exception stack traces. This subset comprises the stack traces related to the main findings of the study (e.g., "undocumented runtime and checked exceptions", and "cross-type wrappings").

5.5 Limitations of the Survey-based Study

Results Generalization. Due to the exploratory nature of the second phase of this work whose goal was to identify the developers' perspectives concerning the exception handling bug hazards found during the repository mining phase, we chose Grounded Theory techniques. The results of our survey-based study may not apply to every Android developer, since other populations might add new insights. The population we collected data from was comprised of Android developers of the GitHub Android projects whose issues were mined in the first phase of this work, and who had time and motivation to answer

[14] code.google.com/p/acra/
[15] https://www.google.com/analytics/
[16] https://bugsnag.com/

the survey questions. Although the themes and findings that emerged in our study cannot be generalized, they give a first view of developers' perspectives about EH bug hazards found. Hence, we believe that the survey-based study has contributed with valuable insights about how Android developers used the exception handling code and what their perspectives regarding exception handling bug hazards are.

Survey Customization. Although the survey was applied to developers who had contributed to at least one of the GitHub apps, analyzed in the mining study, the survey was not customized to each developer. In other words, it did not contain questions focusing on specific bug hazards identified in their apps. That customization of the survey (for each specific developer/app) could give more insights about the exception handling usage and bug hazards, and could even have improved the response rate for the survey. However, we did not followed this approach because the high number of projects analyzed (482 Android apps) and their contributors (1,824 developers) would increase the complexity and time needed (i) to prepare the survey and specially (ii) to analyze the survey responses - since themes could emerge from specific contexts. This approach can be very useful in a guided interview involving some of the respondents to refine the findings of this exploratory survey in a future work.

5.6 Replication Package

All the data used in the mining study and in the survey-based study is publicly available at the ExceptionMiner tool website hosted on GitHub[17]. Specifically we provide: (i) all issues related to Android projects found on GitHub and Google Code used in this study; (ii) all stack traces extracted from issues; (iii) the results of manual inspection steps; (iv) the ExceptionMiner tool we developed to support stack trace extraction and distilling; (v) the survey questions; and (iv) the survey responses and summary.

6 Related Work

In this section, we present work that is related to the present paper, divided into four categories as detailed next.

Analysis and Use of Stack Trace Information. Several papers have investigated the use of stack trace information to support: bug classification and clustering [54, 31, 18], fault prediction models [32], automated bug fixing tools [51] and also the analysis of Android APIs [30]. Kim et al. [31] use an aggregated form of multiple stack traces available in crash reports to detect duplicate crash reports and to predict if a given crash will be fixed. Dhaliwal et al. [18] proposed a crash grouping approach that can reduce bug fixing time by approximately 5%. Wang et al. [54] propose an approach to identify correlated crash types and describe a fault localization method to locate and rank files

[17] `https://github.com/souzacoelho/exceptionminer`

related to the bug described in a stack trace. Schröter et al. [48] conducted an empirical study on the usefulness of stack traces for bug fixing and showed that developers fixed the bugs faster when failing stack traces were included in bug issues. In a similar study, Bettenburg et al. [8] identify stack traces as the second most relevant feature of good bug reports. Sinha et al. [51] proposed an approach that uses stack traces to guide a dataflow analysis for locating and repairing faults that are caused by the implicitly signaled exceptions. Kim at al. [32] proposed an approach to predict the crash-proneness of methods based information extracted from stack traces and methods' bytecode operations. They observed that most of the stack traces were related to NullPointerException and other implicitly thrown exceptions had the higher prevalence in the analyzed set of stacks. Kechagia and Spinellis [30] examined the stack traces embedded in crash reports sent by 1,800 Android apps to a crash report management service (i.e., BugSense). They found that 19% of such stack traces were caused by unchecked and undocumented exceptions thrown by methods defined in the Android API (level 15). Our work differs from Kechagia and Spinellis since it is based on stack traces mined from issues reported by open source developers on GitHub and Google Code. Moreover, our study mapped the origin of each exception (i.e., libraries, the Android platform or the application itself) and investigated the adoption of best practices based on the analysis of stack trace information. Our work also identified the type of each exception mined from issues (classifying them as Error, Runtime or Checked) based on the source code analysis of the exception hierarchy and analyzed the exception wrappings that can happen during the exception propagation. Such analysis revealed intriguing *bug hazards* such as the cross-type exception wrappings not discussed in previous works.

Extracting Stack Traces from natural language artifacts. Apart from issues and bug reports, stack traces can be embedded in other forms of communication between developers, such as discussion logs and emails. Few tools have been proposed to mine stack traces embedded on such resources. Infozilla [9] is based on a set of regular expressions that extract a set of frames related to a stack trace. The main limitation of this solution is that it is not able to extract stack traces embedded in verbose log files (i.e., in which we can find log text mixed with exception frames). Bacchelli et al. [6] propose a solution to recognize stack trace frames from development emails and relate them to code artifacts (i.e. classes) mentioned in the stack trace. In addition to those tools, ExceptionMiner is able to both extract stack traces from natural language artifacts and to classify them into a set of predefined categories.

Empirical Studies on Exception Handling Defects. Cabral and Marques [13] analyzed the source code of 32 open-source systems, both for Java and .NET. They observed that the actions inside handlers were very simple (e.g., logging and presenting a message to the user). Coelho et al. [15] performed an empirical study considering the fault-proneness of aspect-oriented implementations for handling exceptions. Two releases of both Java and AspectJ implementations were assessed as part of that study. Based on the use of an exception flow analysis tool, the study revealed that the AOP refactoring increased the

number of uncaught exceptions, degrading the robustness of the AO version of every analyzed system. The main limitation of approaches based on static analysis approaches are the number of false positives they can generate, and the problems faced when dealing with reflection libraries and dynamic class loading. Pingyu and Elbaum [58] were the first to perform an empirical investigation of issues, related to exception-related bugs, in Android projects. They perform a small scale study in which they manually inspected the issues of 5 Android applications. They observed that 29% had to do with poor exceptional handling code, and this empirical study was used to motivate the development of a tool aiming at amplifying existing tests to validate exception handling code associated with external resources. This work inspired ours, which automatically mined the exception stack traces embedded in issues reported in 639 open source Android projects. The goal of our study was to identify common *bug hazards* in the exception related code that can lead to failures such as uncaught exceptions.

Empirical studies using Android apps. Ruiz et al. [46] investigated the degree of reuse across applications in the Android Market; the study showed that almost 23% of the classes inherited from a base class in the Android API, and that 217 mobile apps were reused completely by another mobile app. Pathak et al. [43] analyzed bug reports and developers' discussions of the Android platform and found that approximately 20% of energy-related bugs in Android occurred after an OS update. McDonnell et al. [39] conducted a case study of the co-evolution behavior of the Android API and 10 dependent applications using the version history data found in GitHub. The study found that approximately 25% of all methods in the client code used the Android API, and that the methods reusing fast-evolving APIs were more defect prone than others. Vasquez et al. [35] analyzed approximately 7K free Android apps and observed that the least successful apps used Android APIs that were on average 300% more change-prone than the APIs used by the most successful apps. Our work differs from the others as it aims at distilling stack trace information from bug reports and combining such information with bytecode analysis, source code analysis and manual inspections to identify *bug hazards* in the exception handling code of Android apps.

Exploratory Survey Studies. Exploratory surveys have been used in the software engineering context to discover the user perspective regarding a broad range of topics such as: assessing the developers' perceptions on productivity [40]; how GitHub developers use pull-requests [26]; how the testing culture of open-source projects can be characterized [44]; and even how developers use Twitter [50]. This kind of study is important as a way of better understanding the developers' behavior and hence providing recommendations and tools to help with specific development tasks.

Exploratory surveys have also targeted Android developers [34, 29, 7, 36]. Kochhar et al. [34] conducted a survey-based study to discover the commonly used tools for mobile app testing as well as the problems faced by developers while testing the apps. The survey was sent to 3,905 emails of Android developers and received 83 responses (response rate of 2.13%). Joorabchi et

al. [29] performed a survey-based study whose goal was to better understand the main challenges developers face when building apps for different mobile devices. They interviewed 12 senior mobile developers and performed a semi-structured survey, with 188 respondents from the mobile development community, since the survey was shared via e-mail groups and social media websites response rate of the survey could not be calculated. Linares-Vásquez et al. [36] surveyed 485 open source Android app and library developers (for projects hosted on GitHub) to understand developers' practices for detecting and fixing performance bottlenecks in mobile apps. This work emailed the survey to 24,340 developers and received 628 responses - the response rate was 2.6%. Bavota et al. [7] investigated the impact of API change-proness and fault-proneness on the user ratings of Android apps. They surveyed developers to assess their perspective on whether such problems could be the cause for unfavorable user ratings. The response rate was 4%. A common characteristic among such works and our work is the low response rate of the surveys (between 4% and 2%). None of these surveys, however, assessed how developers deal with exceptions in Android nor the developers' perspective regarding a set of exception handling bug hazards. In our study, we conduced the first exploratory survey focusing on exception handling issues in Android development.

7 Conclusion

The goal of this paper is two-fold: (i) to investigate to what extent stack trace information can reveal *bug hazards* related to exception handling code that may lead to a decrease in application robustness; and (ii) to assess the developers' perspective concerning the detected exception handling bug hazards.

To realize this goal, we mined the stack traces embedded in all issues defined in 482 Android projects hosted on GitHub and 157 projects hosted on Google Code – overall considering 6,005 exception stack traces. We subsequently surveyed developers associated with a selection of the mined GitHub projects.

Our first key contribution is a novel approach and toolset (ExceptionMiner) for analyzing Java exception stack traces as occurring in GitHub and Google Code issues.

Our second contribution is an empirical study of over 6000 actual stack traces, demonstrating that:

1. Half of the system crashes are due to errors in programming logic, with null pointer exceptions being most prominent;
2. Documentation for explicitly thrown RuntimeExceptions is almost never provided;
3. Extensive use of wrapping leads to hard-to-understand chains violating Java's exception handling principles.

Our third contribution is a qualitative study to assess the developers' perspective concerning the exception handling bug hazards in Android develop-

ment. This study corroborates the findings of our stack trace analysis, most notably the prevalence of null pointer exceptions and the reliability implications of (in particular cross-type) wrappings. Furthermore, we found that few developers are aware of the undocumented checked exceptions signaled by native C code of the Android platform.

In conclusion, our findings shed light on common problems and bug hazards in Java exception handling code, and call for tool support to help developers understand their own and third party exception handling and wrapping logic.

Acknowledgements This work is partially supported by the National Institute of Science and Technology for Software Engineering (INES), CNPq and FACEPE, grants 573964/2008-4, 552645/2011-7, and APQ-1037-1.03/08, CNPq Universal grant 484209/2013-2, and CAPES/PROAP.

References

1. (2013) Checked or unchecked exceptions? `http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html`, online
2. (2014) Java: checked vs unchecked exception explanation. `http://stackoverflow.com/questions/6115896/java-checked-vs-unchecked-exception-explanation`, online
3. (2014) The Java tutorial. Unchecked exceptions: The controversy. `http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html`, online
4. (2014) Null references:the billion dollar mistake, abstract of talk at qcon london. `qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake`, online
5. Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM (2012) Using gui ripping for automated testing of android applications. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, pp 258–261
6. Bacchelli A, Dal Sasso T, D'Ambros M, Lanza M (2012) Content classification of development emails. In: Proceedings of ICSE 2012, pp 375–385
7. Bavota G, Linares-Vasquez M, Bernal-Cardenas CE, Di Penta M, Oliveto R, Poshyvanyk D (2015) The impact of api change-and fault-proneness on the user ratings of android apps. Software Engineering, IEEE Transactions on 41(4):384–407
8. Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008) What makes a good bug report? In: Proceedings of FSE 2008, pp 308–318
9. Bettenburg N, Premraj R, Zimmermann T, Kim S (2008) Extracting structural information from bug reports. In: Proceedings of MSR 2008, ACM, pp 27–30
10. Binder R (2000) Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Professional

11. Bloch J (2008) Effective java. Pearson Education India
12. Brunet J, Guerrero D, Figueiredo J (2009) Design tests: An approach to programmatically check your code against design rules. In: Proceedings of New Ideas and Emerging Research (NIER) track at the International Conference on Software Engineering (ICSE), IEEE, pp 255–258
13. Cabral B, Marques P (2007) Exception handling: A field study in Java and .Net. In: Proceedings of ECOOP 2007, Springer, pp 151–175
14. Charmaz K (2006) Constructing grounded theory: A practical guide through qualitative research. SagePublications Ltd, London
15. Coelho R, Rashid A, Garcia A, Ferrari F, Cacho N, Kulesza U, von Staa A, Lucena C (2008) Assessing the impact of aspects on exception flows: An exploratory study. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, pp 207–234
16. Coelho R, von Staa A, Kulesza U, Rashid A, Lucena C (2011) Unveiling and taming liabilities of aspects in the presence of exceptions: a static analysis based approach. Information Sciences 181(13):2700–2720
17. Csallner C, Smaragdakis Y (2004) Jcrasher: an automatic robustness tester for Java. Software: Practice and Experience 34(11):1025–1050
18. Dhaliwal T, Khomh F, Zou Y (2011) Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In: Proceedings of International Conference on Software Maintenance (ICSM 2011), pp 333–342
19. Enck W, Octeau D, McDaniel P, Chaudhuri S (2011) A study of android application security. In: USENIX security symposium, vol 2, p 2
20. Fraser G, Arcuri A (2013) 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. Empirical Software Engineering pp 1–29
21. Garcia A, Rubira C, et al (2007) Extracting error handling to aspects: A cookbook. In: Proceedings International Conference on Software Maintenance (ICSM), IEEE, pp 134–143
22. Garcia AF, Rubira CM, Romanovsky A, Xu J (2001) A comparative study of exception handling mechanisms for building dependable object-oriented software. Journal of systems and software 59(2):197–222
23. Goodenough JB (1975) Exception handling: issues and a proposed notation. CACM 18(12):683–696
24. Gosling J (2000) The Java language specification. Addison-Wesley Professional
25. Gousios G (2013) The GHTorrent dataset and tool suite. In: Proceedings of the International Working Conference on Mining Software Repositories (MSR), IEEE, pp 233–236
26. Gousios G, Zaidman A, Storey MA, Van Deursen A (2015) Work practices and challenges in pull-based development: the integrator's perspective. Tech. rep.
27. Hindle A, Bird C, Zimmermann T, Nagappan N (2015) Do topics make sense to managers and developers? Empirical Software Engineering 20(2):479–515

28. Jo JW, Chang BM, Yi K, Choe KM (2004) An uncaught exception analysis for java. Journal of systems and software 72(1):59–69
29. Joorabchi ME, Mesbah A, Kruchten P (2013) Real challenges in mobile app development. In: Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on, IEEE, pp 15–24
30. Kechagia M, Spinellis D (2014) Undocumented and unchecked: exceptions that spell trouble. In: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, pp 312–315
31. Kim S, Zimmermann T, Nagappan N (2011) Crash graphs: An aggregated view of multiple crashes to improve crash triage. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, pp 486–493
32. Kim S, Zimmermann T, Premraj R, Bettenburg N, Shivaji S (2013) Predicting method crashes with bytecode operations. In: Proceedings of the 6th India Software Engineering Conference, pp 3–12
33. Ko AJ, DeLine R, Venolia G (2007) Information needs in collocated software development teams. In: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, pp 344–353
34. Kochhar PS, Thung F, Nagappan N, Zimmermann T, Lo D (2015) Understanding the test automation culture of app developers. In: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on, IEEE, pp 1–10
35. Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Di Penta M, Oliveto R, Poshyvanyk D (2013) API change and fault proneness: A threat to the success of Android apps. In: Proceedings of FSE 2013, ACM, pp 477–487, DOI 10.1145/2491411.2491428, URL `http://doi.acm.org/10.1145/2491411.2491428`
36. Linares-Vásquez M, Vendome C, Luo Q, Poshyvanyk D (2015) How developers detect and fix performance bottlenecks in android apps. In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, pp 352–361
37. Maji AK, Arshad FA, Bagchi S, Rellermeyer JS (2012) An empirical study of the robustness of inter-component communication in Android. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, pp 1–12
38. Mandrioli D, Meyer B (1992) Advances in object-oriented software engineering. Prentice-Hall, Inc.
39. McDonnell T, Ray B, Kim M (2013) An empirical study of api stability and adoption in the android ecosystem. In: Proceedings International Conference on Software Maintenance (ICSM), pp 70–79
40. Meyer AN, Fritz T, Murphy GC, Zimmermann T (2014) Software developers' perceptions of productivity. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, pp 19–29
41. Miller R, Tripathi A (1997) Issues with exception handling in object-oriented systems. In: Proceedings of ECOOP'97, Springer, pp 85–103

42. Nanda MG, Sinha S (2009) Accurate interprocedural null-dereference analysis for java. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, IEEE, pp 133–143
43. Pathak A, Hu YC, Zhang M (2011) Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In: Proceedings of the 10th ACM Workshop on Hot Topics in Networks, ACM, New York, NY, USA, HotNets-X, pp 5:1–5:6, DOI 10.1145/2070562.2070567, URL http://doi.acm.org/10.1145/2070562.2070567
44. Pham R, Singer L, Liskin O, Figueira Filho F, Schneider K (2013) Creating a shared understanding of testing culture on a social coding site. In: Software Engineering (ICSE), 2013 35th International Conference on, IEEE, pp 112–121
45. Robillard MP, Murphy GC (2000) Designing robust Java programs with exceptions. In: Proceedings International Conference on the Foundations of Software Engineering (FSE), pp 2–10
46. Ruiz I, Nagappan M, Adams B, Hassan A (2012) Understanding reuse in the Android market. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp 113–122, DOI 10.1109/ICPC.2012.6240477
47. Sacramento P, Cabral B, Marques P (2006) Unchecked exceptions: can the programmer be trusted to document exceptions. In: International Conference on Innovative Views of .NET Technologies
48. Schröter A, Bettenburg N, Premraj R (2010) Do stack traces help developers fix bugs? In: Proceedings Working Conference on Mining Software Repositories (MSR), IEEE, pp 118–121
49. Shah HB, Gorg C, Harrold MJ (2010) Understanding exception handling: Viewpoints of novices and experts. IEEE Trans Soft Eng 36(2):150–161
50. Singer L, Figueira Filho F, Storey MA (2014) Software engineering at the speed of light: how developers stay current using twitter. In: Proceedings of the 36th International Conference on Software Engineering, ACM, pp 211–221
51. Sinha S, Shah H, Görg C, Jiang S, Kim M, Harrold MJ (2009) Fault localization and repair for Java runtime exceptions. In: Proceedings International Symposium on Software Testing and Analysis (ISSTA), ACM, pp 153–164
52. Van Dooren M, Steegmans E (2005) Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. ACM SIGPLAN Notices 40(10):455–471
53. Wadler P (1995) Monads for functional programming. In: Advanced Functional Programming, Springer, pp 24–52
54. Wang S, Khomh F, Zou Y (2013) Improving bug localization using correlations in crash reports. In: Proceedings Working Conference on Mining Software Repositories (MSR 2013), ACM/IEEE, pp 247–256
55. Wasserman AI (2010) Software engineering issues for mobile application development. In: Proceedings of the FSE/SDP workshop on Future of software engineering research, ACM, pp 397–400

56. Wirfs-Brock RJ (2006) Toward exception-handling best practices and patterns. Software, IEEE 23(5):11–13
57. Yuan D, Luo Y, Zhuang X, Rodrigues GR, Zhao X, Zhang Y, Jain P, Stumm M (2014) Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014., pp 249–265
58. Zhang P, Elbaum S (2012) Amplifying tests to validate exception handling code. In: Proceedings International Conference on Software Engineering (ICSE), IEEE Press, Piscataway, NJ, USA, pp 595–605, URL `http://dl.acm.org/citation.cfm?id=2337223.2337293`

TUD-SERG-2016-018
ISSN 1872-5392