Unveiling Exception Handling Bug Hazards in Android based on GitHub and Google Code Issues

Roberta Coelho*, Lucas Almeida*, Georgios Gousios†, Arie van Deursen‡
*Federal University of Rio Grande do Norte
Natal, Brazil
Email: roberta@dimap.ufrn.br,lucas.almeida@ppgsc.ufrn.br
†Radboud University Nijmegen
Nijmegen, The Netherlands
Email: georgios@cs.ru.nl
‡Delft University of Technology
Delft, The Netherlands
Email: arie.vandeursen@tudelft.nl

***Abstract*—This paper reports on a study mining the exception stack traces included in 159,048 issues reported on Android projects hosted in GitHub (482 projects) and Google Code (157 projects). The goal of this study is to investigate whether stack trace information can reveal *bug hazards* related to exception handling code that may lead to a decrease in application robustness. Overall 6,005 exception stack traces were extracted, and subjected to source code and bytecode analysis. The outcomes of this study include the identification of the following *bug hazards*: (i) unexpected cross-type exception wrappings (for instance, trying to handle an instance of OutOfMemoryError "hidden" in a checked exception) which can make the exception-related code more complex and negatively impact the application robustness; (ii) undocumented runtime exceptions thrown by both the Android platform and third party libraries; and (iii) undocumented checked exceptions thrown by the Android Platform. Such undocumented exceptions make difficult, and most of the times infeasible for the client code to protect against "unforeseen" situations that may happen while calling third-party code. This study provides further insights on such *bug hazards* and the robustness threats they impose to Android apps as well as to other systems based on the Java exception model.**

I. Introduction

In recent years, we have witnessed an astonishing increase in the number of mobile applications. These applications extend phones capabilities far beyond basic calls and textual messages. They must, however, face an increasing number of threats to application robustness arising from failures in the underlying middleware and hardware (e.g., camera, sensors); failures in third party services and libraries; compatibility issues [36]; memory and battery restrictions; and noisy external resources [52] (e.g., wireless connections,GPS, bluetooth).

Therefore, techniques for error detection and handling are not an optional add-on but a fundamental part of such apps. The exception handling mechanism [26], 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.

Studies have shown that exception-related code is generally poorly understood and among the least tested parts of the system [37], [40], [44], [24], [25], [17], [19], [51]. As a consequence they may inadvertently negatively affect the system: exception-related code may introduce failures such as uncaught exceptions [29], [52] - which can lead to system crashes, making the system even less robust [19].

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 on a log file [27]. 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 [27], [15].

This study performs a post mortem analysis of the exception stack traces included in issues reported on Android projects hosted in 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* [14] is a circumstance that increases the chance of a bug to be 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.

To guide this investigation we compiled general guidelines on how to use Java exceptions proposed by Gosling [27], Wirfs-Brock [50] and Bloch [15]. Then, using a custom tool called ExceptionMiner, which we developed specifically for this study, we mine stack traces from the issues reported on 482 Android projects hosted in GitHub and 157 projects hosted in 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 on 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 high prevalence of NullPointerExceptions is in line with findings of earlier research [32], [23], [20], as are the undocumented runtime exceptions signaled by the Android Platform [30]. Some of the findings of our study emphasize the impact of these bug hazards on the application robustness by mining a different source of information 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.

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 on languages and tools to better support exception handling in Android and Java environments.

The remainder of this paper is organized as follows. Section II provides the necessary background on the Android platform and Java exception model. Section III presents the study design. Section III-C describes the ExceptionMiner tool we developed to conduct our study. Section IV reports the findings of our study. Section V provides a discussion of the wider implications of our results. Section VI presents the threats to validity associated to this study. Finally Section VII describes related work, and Section VIII concludes the paper and outlines directions for future work.

II. Background

A. 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.

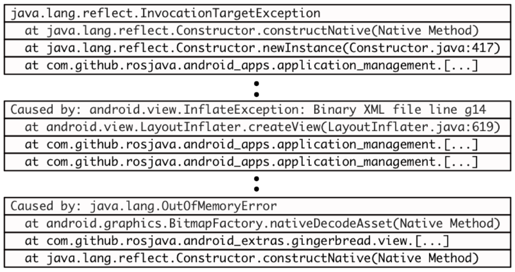
B. Exception Model in Java

Exception Types. In Java, exceptions are represented according to a class hierarchy, on 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) [27]. Checked exception received their name because they must be declared on 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 on 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 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 [27] 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 [8], [6], [1] Section II-C 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 [37].



```
java.lang.reflect.InvocationTargetException
  at java.lang.reflect.Constructor.constructNative(Native Method)
  at java.lang.reflect.Constructor.newInstance(Constructor.java:417)
  at com.github.rosjava.android_apps.application_management.[...]
                              :
Caused by: android.view.InflateException: Binary XML file line g14
  at android.view.LayoutInflater.createView(LayoutInflater.java:619)
  at com.github.rosjava.android_apps.application_management.[...]
  at com.github.rosjava.android_apps.application_management.[...]
                              :
Caused by: java.lang.OutOfMemoryError
  at android.graphics.BitmapFactory.nativeDecodeAsset(Native Method)
  at com.github.rosjava.android_extras.gingerbread.view.[...]
  at java.lang.reflect.Constructor.constructNative(Native Method)
```

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

A common way of propagating exceptions in Java programs is through exception wrapping (also called chaining): One exception is caught and wrapped in another 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 as just stack trace. The bottom part of the stack trace is the *root exception*, which indicates the first reason (root cause) for the exception thrown (in this case, the computer run 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). The execution flow between the root exception and the wrapper may include other intermediate exception wrappers. In all levels, the exception *signaler*, is the method that threw the exception, represented on the stack trace as the first method call below the exception declaration.

C. Best Practices

Several general guidelines have been proposed on how to use Java exceptions [35], [27], [50], [15]. 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 ([35], [27], [50], [15]). 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 [27].

II-Error represents an unrecoverable condition which should not be handled ([27]). 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 ([27], [15]). 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. ([35], [27], [50], [15]). For checked exceptions, this is automatically the case. Bloch [15] 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 [40], [50].

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 I: Labels on issues including exception stack traces.

III. Study Design

The goal of our mining study is to explore to what extent exception stack traces contained in issues of Android projects (hosted in GitHub and Google Code), can reveal *bug hazards* in the exception-related code of both the applications and the underlying 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.

This exploration is guided by the set of best practices covered in Section II-C. To support our investigation, we developed a tool called ExceptionMiner (Section III-C) which extracts the exception stack traces embedded on 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 III-D). 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 on Android projects hosted on GitHub (1) and Google Code (2) are recovered. Then the stack traces embedded 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 on 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 codes are generally not publicly available. Open source Android open-source apps have also been the target of other research [33], [41] addressing the *reuse* and API stability.

A. Android Apps in GitHub

This study uses the dataset provided by the GHTorrent project [28], an off-line mirror of the data offered through the Github API. To identify Android projects, we performed a case

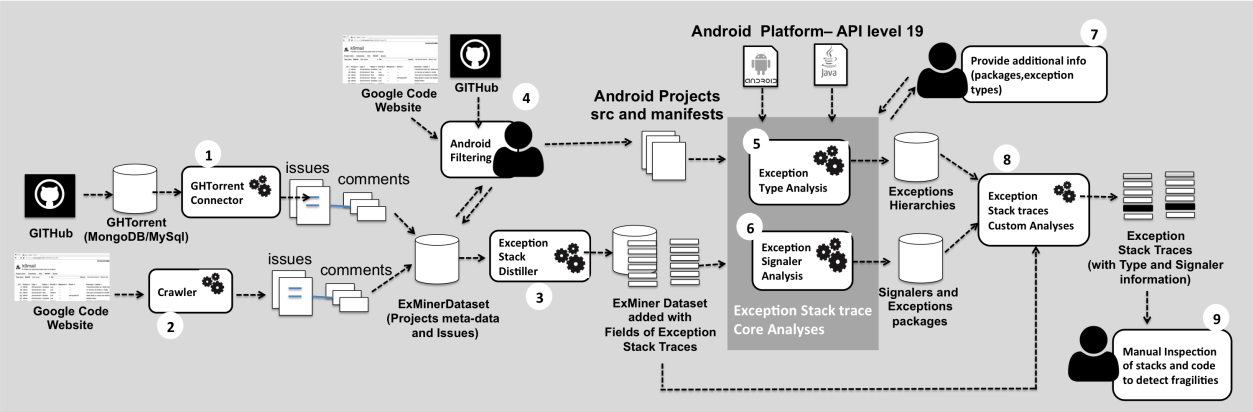

Android Platform– API level 19
7
Provide additional info (packages,exception types)
Google Code Website
GITHub
4
Android Projects src and manifests
Android Filtering
1
GHTorrent Connector
issues
comments
GITHub
GHTorrent (MongoDB/MySql)
5
Exception Type Analysis
Exceptions Hierarchies
8
Exception Stack traces Custom Analyses
Exception Stack Traces (with Type and Signaler information)
6
Exception Signaler Analysis
Signalers and Exceptions packages
Exception Stack trace Core Analyses
Google Code Website
Crawler
2
issues
comments
ExMinerDataset (Projects meta-data and Issues)
Exception Stack Distiller
3
ExMiner Dataset added with Fields of Exception Stack Traces
9
Manual Inspection of stacks and code to detect fragilities

Fig. 2: Study overview.

insensitive search for the term “android” in the repository’s names and short descriptions. Up to 23 February 2014, when we queried GHTorrent, this resulted in 2,542 repositories. Running the ExceptionMiner tool in 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 106 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 overall 31,592 issues from which 4,042 exception stack traces were extracted.

Issues on Github are different from issues on 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, on which 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 I illustrates the ocurrences 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* on the exception-related code. Because of this, we opted for not restricting the analysis to just defect issues.

B. 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 access the information related to hosted projects.[2] To overcome this limitation we needed to implement a Web Crawler (incorporated in ExceptionMiner tool described next) that navigates through 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 in Google Code, we performed a similar heuristic: we performed a case insensitive search (on the Google Code search interface) for the term “android”. On 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 in 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 I illustrates the occurrences of different labels on the issues including exception stack traces. Differently from GitHub, in 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 its labels).

C. 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 GHTorrent database, and a Google Code connector which is comprised of a Web Crawler that can traverse the Google Code web interface and extract the 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 is based on a combination between a parser (based on regular ex-

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

[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”.

pressions) 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 associated to 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 on 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 [16] to inspect the bytecode of the exceptions reported on 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 exception was available on the project or on a reused Java library. An 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 on stack traces that were defined on 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 II 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 Plaform; the Android Libcore; and each analyzed Application.

To discover the packages for the first two origins we can use to 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 appm filtering 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 on core libraries, nor on applications. Table II summarizes this signaler classification.

Signaler	Description
android	If the exception is thrown in a method defined in Android platform.
app	If the exception is thrown in an method defined on 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 on a method that was not defined by any of the elements above.

TABLE II: Sources of exceptions in Android

D. 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 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 be automatically identified by the ExceptionMiner tool (because they were defined on previous versions of libraries, apps and Android Plaform). When the exception could not be found automatically or manually (because they were defined on 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 V).

E. Replication Package

All the data used in this study is publicly available at https://github.com/souzacoelho/exceptionminer. Specifically we provide: (i) all issues related to Android projects found in 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.

IV. The Study Results

This section presents the results of the study, providing both a quantitative and qualitative analysis of the outcomes. We center our presentation of the results around *bug hazards* we distilled from (1) common root exceptions; (2) exception types; and (3) exception wrappings.

A. Common Root Exceptions

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

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

[3] In several exception stack traces, the exception frames were preceeded 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.

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 III: Root Exceptions occurrences and popularity in repositories hosted in Google Code (GC) and GitHub(GH).

projects reported at least one exception stack trace on which the NullPoiterException 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 on an instance of IllegalArgumentException, while others [15] argue that the NullPointerException makes the cause of the problem explicit and hence can be signalled by an API expecting a non-null argument.

The high prevalence of NullPointerException is aligned with the findings of other research [32], [23], [20], [30]. For instance, Sunghun et al. [32] show that 38% the bugs related to exception handling in the Eclipse project are caused by NullPointerException. Furhtermore, Kechagia and Spinellis showed that the NullPointerException was the most reported exception on the crash reports sent to BugSense (a bug report management service for Android applications) [30]. Other research on robustness testing [34], [20] 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 some concerns that are known as sources of faults in mobile development: concurrency [10] backward compatibility [36], security [22], [49] and resource management (IO, Memory, Batery) [52].

Since it is infeasible to inspect the code responsible for throwning every exception reported in this study, the concern identification of each exception was based on intended meaning of the particular exception type, as defined in its Javadoc documentaion 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 [7], such as InstantiationError, VerifyError, and IllegalAccessError.

Concern	**% Occurrences on stacks**
Programming logic (java.lang and util)	52,0%
Resources (IO, Memory, Batery)	23,9%
Security	4,1%
Concurrency	2,9%
Backward compatibility	5,5%
Specific Exceptions	4,9%
General (Error, Exception, Runtime)	6,7%

TABLE IV: Identifying the concerns related to root exceptions

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 releated to each root exception. Table IV 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%.

More than 50% of the uncaught exceptions are due to errors in programming logic, with the NullPointerException as most prevalent exception. For another 25% the root cause relates to resource constraints.

B. *Exception Types*

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 related to it in the stack traces (Section III-C). Table V 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 on the Application (47.3%) or on the Android platform (34.3%). We could also find runtime exceptions thrown by library code (17.7%).

We can also see, from Table V, that in contrast with the other origins, most of the exceptions signaled on Android

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 V: Types and origins of root exceptions.

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 II-C) because by using checked exceptions a library can define a precise *exception interface* [37] to its clients. Since such libraries are widely used in several projects, this finding can be attributed to the libraries' maturity.

Almost two thirds of all crashes come from run time exceptions. Most 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 libraries/framework reusable methods. To investigate the conformance to this practice, we firstly filtered out all the exceptions implicitly signaled by the runtime environment (due to programming mistakes) - since these exceptions should not be documented on the method signature. Then we inspected the code for each method (defined either in the Android Aplication Framework or in third-party libraries) explicitly signaling a runtime exception.

Table VI 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 on the *exception interface* (i.e., using throws clause in the method signature). From this set only one method (defined on 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 [42], [30]. Sacramento et al [42] observed that the runtime exceptions in .NET programs are most often not documented. Kechagia and Spinellis [30] identified a set of methods on the Android API which do not document its 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 on Java semantic 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.

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

TABLE VI: Absence of exception interfaces on methods.

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 Exceptino
7	java.lang.ExceptionInInitializerError - java.lang.NullPointerException
8	java.lang.ExceptionInInitializerError - java.lang.IllegalArgumentException

TABLE VII: Examples of Cross-type wrappings

Only a small fraction (4%, 267 stack traces) of run time exceptions are programmatically thrown. Almost none (0.4%, just one) of these were documented.

Missing Checked Exceptions on Exception Interfaces. Our exception stack trace analysis revealed an unexpected bug hazard: a checked exception thrown by a native method and not declared on the exception interface of these 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 to the native implementations. Consequently, since it is not expected by the programmer, when such method throws this exception, such an undocumented exception may remain uncaught and cause the app to crash, or maybe mistakenly handled by subsumption. The exception stack traces reporting this scenario actually correspond to a real bug of 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 on the exception interface.

C. Exception Wrappings

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 VIII presents some wrappings found in

Wrapper Type	Root Cause Type	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 VIII: Wrappings comprising different exception types.

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 an Error. From Table VIII, 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. Table VII presents examples of exceptions that were actually wrapped in this code snipet. Such wrappings mask an unrecoverable Error into a general runtime exception.

```
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);
}
```

Runtime Exception wrapping a Checked Exception. This wrapping was responsible for 49.5% of the cross-type wrappings. From this set 50% were performed on methods defined on Android platform. We observe that it is a common implementation practice in the methods of Android platform. However, using such a general exception, is considered a bad practice according to the Java specification and common guidelines, as it loses contextual information about the exception.

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 VII illustrates some of these wrappings some of them are masking an OutOfMemoryError into a checked exception. Such wrappings may also mask an unrecoverable error and may lead to "exception confusion" described next.

Error wrapping Runtime and Checked Exceptions Table VII illustrates examples of instances of Error wrapping instances of RuntimeException. Although such a wapping mixes different exception types, since there is no obligation associated to 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 on the point where the class is first used. Table VII also illustrates examples of such wrappings. Although such a wrapping may represent a design decision, it violates the best practice related to checked exceptions and errors as it mixes the intended handling behaviour associated to 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 from design decisions, the mis-use of exception wrapppings 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 Java Specification and best practices). However, such 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 violate the semantics of Java's original exception design (e.g., when mapping unrecoverable Errors to other types of exceptions), and may lead to lengthy exception chains.

V. Discussion

> *"Everybody hates thinking about exceptions, because they are not supposed to happen"*
> *(Brian Foote)*[4]

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 the 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. Furthermore, this calls for empirical studies on the actual usefulness of Java's hybrid exception model.

On the null pointer problem. The null reference was firstly introduced by Tony Hoare in ALGOL W, which after some years he called his "one-billion-dollar mistake" [9]. In

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

this sudy, the null references were, infact, responsible for several reported issues - providing further evidence to Hoare's statement. This observation emphasizes the need for solutions to avoid NullPointerExceptions, such as: (i) lightweight intra-method null pointer analysis as supported by Java 8 @Nullable annotations[5]; (ii) inter-method null pointer analysis tools such as the one proposed by Nanda and Sinha [38]; or (iii) language designs which avoid null pointers, such as Monads [47] (as used in functional languages for values that may not be available or computations that may fail) could improve the robustness of Java programs.

Preventing uncaught exceptions In this study we could observe undocumented runtime exceptions thrown by third party code, and even undocumented checked exception 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 a library code.

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, such a general handler will not have enough contextual information to adequately handle the exception, beyond storing a message in a log file and restarting the application. However, such a handler cannot replace a carefully designed exception handling policy [40], 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 [46].

VI. 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 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 developers classified a randomly selected sample (of 25 exception types from the total of 100); the interater agreement was high (96%).

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* on 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 imposed to other empirical studies which also use free or open-source Android apps [33], [36], [41]. Moreover, several exception stack traces that support the findings of this study refered to exceptions coming from methods defined on Android Application Framework and third-party libraries. Additionaly, the *bug hazards* observed in this study are due to characteristics of Java exception model, which can impose challenges the robustness of not only to 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 on issues reported on Github and Googlecode projects. In using these datasets, we make an underlying assumption: the stack traces reported on issues are representative of valid crash information of the applications. One way to mitigate this threat would be to access to the full set of crash data per application. Although some services exist to collect crash data from mobile applications [3], [4], [5], [2], 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 to 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").

VII. Related Work

In this section, we present work that is related to the present paper, divided into four categories: i) papers that use the information available on stack traces; ii) empirical studies on the usage of Java Exceptions and its fault proneness; and iii) tools to extract stack trace information from natural language artifacts (e.g., issues and emails) and iv) empirical studies involving Android apps.

Analysis and Use of Stack Trace Information. Several papers have investigated the use of stack trace information to support: bug classification and clustering [48], [31], [21], fault prediction models [32], automated bug fixing tools [45] 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. [21] proposed a crash grouping approach that can reduce bug fixing time in approximately 5%. Wang et al. [48] propose an approach to identify correlated crash types and describe a fault localization method to locate and rank files related to the bug described on a stack trace. Schroter et al. [43] 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 on bug issues. In a similar study, Bettenburg et al. [12] identify stack traces as the second most stack trace feature for developers. Sinha et al. [45] proposed an approach that uses stack traces to guide a dataflow

[5] 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.

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 on the analyzed set of stacks. Kechagia and Spinellis [30] examined the stack traces embedded on 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 on Android API (level 15). Our work differs from Kechagia and Spinellis since it is not based on stack traces mined from issues reported by open source developers on GitHub and Googlecode. 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 analized the exception wrappings that can happen during the exception propagation. Such analysis revealed intrigging *bug hazards* such as the cross-type exception wrappings not discussed in previous works.

Empirical Studies on Exception Handling Defects. Cabral and Marques [17] 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 present a message to the user). Coelho et al. [18] 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 static analysis based approaches are the number of false positives they can generate, and the problems the faced when dealing with reflection libraries and dynamic class loading. Pingyu and Elbaum [52] were the first to perform an empirical investigation of issues, related to exception-related bugs, on Android projects. They perform a small scale study on which they manually inspected the issues of 5 Android applications. They observed that 29% had to do with poor exceptional handling code, 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 automaticaly mined the exception stack traces embedded on issues reported on 639 open source Android projects. The goal of our study was to identify common *bug hazards* on the exception related code that can lead to failures such as uncaught exceptions.

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 [13] 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 on verbose log files (i.e., on which we can find log text mixed with exception frames). Bacchelli et al. [11] propose a solution to recognize stack trace frames from development emails and relate it to code artifacts (i.e. classes) mentioned on the stack trace. In addition to those tools, ExceptionMiner is able to both extract stack traces from natural language artifacts and to classify them in a set of predefined categories.

Empirical studies using Android apps. Ruiz et al. [41] investigated the degree of reuse across applications in 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. [39] analyzed bug reports and developers discussions of Android platform and found out that approximately 20% of energy-related bugs in Android occurred after an OS update. McDonnell et al. [36] conducted a case study of the co-evolution behavior of 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 then others. Vsquez et al. [33] analyzed approximately 7K free Android apps and observed that the last 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 of bug reports and combine such information with bytecode analysis, source code analysis and manual inspections to identify *bug hazards* on the exception handling code of Android apps.

VIII. Conclusion

The goal of this paper is 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. To that end, we mined the stack traces embedded in all issues defined in 482 Android projects hosted in Github and 157 projects hosted in Google Code. Overall it included 6,005 exception stack traces.

Our first key contribution is a novel approach and toolset (ExceptionMiner) for analyzing Java exception stack traces as occurring on 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 results 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.

References

[1] Checked or unchecked exceptions? http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html, Oct 2013. Online.

[2] Acra. code.google.com/p/acra/, Mar 2014. Online.

[3] Bugsense. https://www.bugsense.com/, Mar 2014. Online.

[4] Bugsnag. https://bugsnag.com/, Mar 2014. Online.

[5] Google analytics. https://www.google.com/analytics/, Mar 2014. Online.

[6] Java: checked vs unchecked exception explanation. http://stackoverflow.com/questions/6115896/java-checked-vs-unchecked-exception-explanation, Mar 2014. Online.

[7] Java specification on backward compatibility. http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html, Jan 2014. Online.

[8] The Java tutorial. Unchecked exceptions: The controversy. http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html, Mar 2014. Online.

[9] Null references: The billion dollar mistake, abstract of talk at qcon londo. qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake, Mar 2014. Online.

[10] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[11] Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. Content classification of development emails. In *Proceedings of ICSE 2012*, pages 375–385, 2012.

[12] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of FSE 2008*, pages 308–318, 2008.

[13] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *Proceedings of MSR 2008*, pages 27–30. ACM, 2008.

[14] Robert Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[15] Joshua Bloch. *Effective java*. Pearson Education India, 2008.

[16] J. Brunet, D. Guerrero, and J. Figueiredo. 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)*, pages 255–258. IEEE, 2009.

[17] Bruno Cabral and Paulo Marques. Exception handling: A field study in Java and .Net. In *Proceedings of ECOOP 2007*, pages 151–175. Springer, 2007.

[18] Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Ferrari, Nélio Cacho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 207–234. Springer-Verlag, 2008.

[19] Roberta Coelho, Arndt von Staa, Uirá Kulesza, Awais Rashid, and Carlos Lucena. Unveiling and taming liabilities of aspects in the presence of exceptions: a static analysis based approach. *Information Sciences*, 181(13):2700–2720, 2011.

[20] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[21] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In *Proceedings of International Conference on Software Maintenance (ICSM 2011)*, pages 333–342, 2011.

[22] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.

[23] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, pages 1–29, 2013.

[24] A Garcia, CMF Rubira, et al. Extracting error handling to aspects: A cookbook. In *Proceedings International Conference on Software Maintenance (ICSM)*, pages 134–143. IEEE, 2007.

[25] Alessandro F Garcia, Cecılia MF Rubira, Alexander Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of systems and software*, 59(2):197–222, 2001.

[26] John B Goodenough. Exception handling: issues and a proposed notation. *CACM*, 18(12):683–696, 1975.

[27] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[28] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, pages 233–236. IEEE, 2013.

[29] Jang-Wu Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. An uncaught exception analysis for java. *Journal of systems and software*, 72(1):59–69, 2004.

[30] Maria Kechagia and Diomidis Spinellis. Undocumented and unchecked: exceptions that spell trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 312–315. ACM, 2014.

[31] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. 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)*, pages 486–493. IEEE, 2011.

[32] Sunghun Kim, Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, and Shivkumar Shivaji. Predicting method crashes with bytecode operations. In *Proceedings of the 6th India Software Engineering Conference*, pages 3–12, 2013.

[33] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. API change and fault proneness: A threat to the success of Android apps. In *Proceedings of FSE 2013*, pages 477–487. ACM, 2013.

[34] Amiya Kumar Maji, Fahad A Arshad, Saurabh Bagchi, and Jan S Rellermeyer. 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)*, pages 1–12. IEEE, 2012.

[35] Dino Mandrioli and Bertrand Meyer. *Advances in object-oriented software engineering*. Prentice-Hall, Inc., 1992.

[36] T. McDonnell, B. Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Proceedings International Conference on Software Maintenance (ICSM)*, pages 70–79, 2013.

[37] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *Proceedings of ECOOP'97*, pages 85–103. Springer, 1997.

[38] Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for java. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 133–143. IEEE, 2009.

[39] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 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*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[40] Martin P. Robillard and Gail C. Murphy. Designing robust Java programs with exceptions. In *Proceedings International Conference on the Foundations of Software Engineering (FSE)*, pages 2–10, 2000.

[41] I.J.M. Ruiz, M. Nagappan, B. Adams, and A.E. Hassan. Understanding reuse in the Android market. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 113–122, 2012.

[42] Paulo Sacramento, Bruno Cabral, and Paulo Marques. Unchecked exceptions: can the programmer be trusted to document exceptions. In *International Conference on Innovative Views of .NET Technologies*, 2006.

[43] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *Proceedings Working Conference on Mining Software Repositories (MSR)*, pages 118–121. IEEE, 2010.

[44] Hina B Shah, Carsten Gorg, and Mary Jean Harrold. Understanding exception handling: Viewpoints of novices and experts. *IEEE Trans. Soft. Eng.*, 36(2):150–161, 2010.

[45] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for Java runtime

exceptions. In *Proceedings International Symposium on Software Testing and Analysis (ISSTA)*, pages 153–164. ACM, 2009.

[46] Marko Van Dooren and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. *ACM SIGPLAN Notices*, 40(10):455–471, 2005.

[47] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[48] Shaohua Wang, Foutse Khomh, and Ying Zou. Improving bug localization using correlations in crash reports. In *Proceedings Working Conference on Mining Software Repositories (MSR 2013)*, pages 247–256. ACM/IEEE, 2013.

[49] Anthony I Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 397–400. ACM, 2010.

[50] Rebecca J Wirfs-Brock. Toward exception-handling best practices and patterns. *Software, IEEE*, 23(5):11–13, 2006.

[51] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. 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.*, pages 249–265, 2014.

[52] Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code. In *Proceedings International Conference on Software Engineering (ICSE)*, pages 595–605, Piscataway, NJ, USA, 2012. IEEE Press.