Improving the quality of APIs through the analysis of software crash reports

Maria Kechagia, Dimitris Mitropoulos, and Diomidis Spinellis

Athens University of Economics and Business
Department of Management Science and Technology
`{mkechagia, dimitro, dds}@aueb.gr`

Abstract

Modern programs depend on APIs to implement a significant part of their functionality. Apart from the way developers use APIs to build their software, the stability of these programs relies on the APIs design and implementation. In this work, we evaluate the reliability of APIs, by examining software telemetry data, in the form of stack traces, coming from Android application crashes. We got 4.9 GB worth of crash data that thousands of applications send to a centralized crash report management service. We processed that data to extract approximately a million stack traces, stitching together parts of chained exceptions, and established heuristic rules to draw the border between applications and API calls. We examined 80% of the stack traces to map the space of the most common application failure reasons. Our findings show that the top ones can be attributed to memory exhaustion, race conditions or deadlocks, and missing or corrupt resources. At the same time, a significant number of our stack traces (over 10%) remains unclassified due to generic unchecked exceptions, which do not highlight the problems that lead to crashes. Finally, given the classes of crash causes we found, we argue that API design and implementation improvements, such as specific exceptions, non-blocking algorithms, and default resources, can eliminate common failures.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, D.2.5 Testing and Debugging, D.2.7 Distribution, Maintenance, and Enhancement

Keywords and phrases application programming interfaces, mobile applications, crash reports, stack traces

Digital Object Identifier 10.4230/OASIcs.ICCSW.2013.57

1 Introduction

Many modern applications use Application Programming Interfaces (APIs) to build their basic functionalities. The stability of these applications depends not only on the use of the APIs by developers, but, also, on the API design and implementation itself.

Even though the software engineering literature encounters works related to software development practices [7], metrics [6], and bug report analysis [10], there are limited studies regarding APIs. We have mainly found sources regarding the usability of APIs [8], [9] and general design practices [4], [2]. Therefore, there is a demand for studies on the assessment of APIs.

In this work, we report how we used software telemetry data, in the form of stack traces, coming from Android application crashes, to analyze their causes and evaluate the reliability of the used APIs. First, we got a 4.9 GB data dump of crash reports from several mobile applications. Then, we processed that data to get an amount of a million Java stack traces in an appropriate form for our analysis. Finally, we applied heuristic rules to draw the border between applications and API calls. This helped us to locate problematic calls to API methods and investigate the reasons that these API deficiencies lead to applications crashes. In addition, knowing the crash causes map of our sample's applications, we were able to argue about related API design and implementation recommendations.

We chose to focus on the study of API crashes for a number of reasons. First, crashes that could have been avoided through a better API land on the hands of application builders. These builders can fix their applications on a case-by-case basis. Thus, locating weaknesses in the APIs and improving their design or implementation can ensure the stability of the thousands of applications that use them. Finally, the fact that most APIs are available as open source software makes them a valuable ground for research.

In addition, we chose the Android platform as the subject of our study because of its popularity, diversity, and availability. Specifically, more than 800 million devices use the Android platform and 700,000 applications are written for Android. In addition, the Android API is quite large (3,000 classes and 300 packages) for examination and its interfaces are open source.

In the rest of this work, we first outline the methods we used (Section 2). In Section 3, we discuss the crash categories we found, and make API recommendations. In Section 4, we present the threats to validity of our study, and we end up with our conclusions and future work in Section 5.

2 Methodology

Our methodology involves data collection, cleaning, processing, and analysis. First, we got our data set and we conformed the stack traces to a certain format for analysis. Then, we applied heuristic rules to the stack traces to extract from each a representative triplet—**signature** hereafter, related to the crash cause. We sorted the signatures based on the times they appear in the stack traces, and we examined the top 600 ones (80% of the total population) to investigate the reasons behind the application failures. Finally, we categorized the crash causes we found into main classes and we made related API recommendations.

2.1 Data Origin

The subject of our empirical study consists of Java stack traces, coming from Android application crashes, collected through a centralized crash report management service.

Android mobile phones are embedded devices that use the Linux operating system and host applications. Here, we briefly discuss an overview of the Android framework. In the bottom layer, there is the Linux kernel, which is the border between the device and the software. It provides services such as memory management, networking, and power management. In the middle layer, there is the Dalvik process virtual machine (VM) for the running of several applications on the system and the Java Native Interface (JNI) that is used to perform calls from Java code into native code. Finally, on the top layer, there are several Java classes coming from: 1) basic applications (contacts, browser, phone), 2) third-party applications, and 3) the Java Platform (J2SE). The methods of these classes are used for the development of Android applications and consist subject of our study.

The provider of our data set is the BugSense Inc., a privately held company, founded in 2011, and based in San Francisco. The aim of BugSense is to provide error reports and analytics regarding the performance and quality of mobile applications. Our sample comes to 2,042,700 crash reports, collected in real time from the 13th of January of 2012 to the 11th of April of 2012, from 4,618 distinct applications. The examined Android API refers to versions from 1.0.0 to 4.1.1.

2.2 Data Cleaning

In order to conduct our analysis, we first needed to clean our data. From our initial sample, we only kept Java stack traces from Android applications. For this, we wrote a program in Python and parsed our data set. Specifically, by using regular expressions, we checked the format validity of the stack traces, based on the `printStackTrace()` method, from the `Throwable`[1] Java class. Thus, from our initial sample, we concluded on 901,274 well-formed Java stack traces. Listing 1 shows a representative example from the `Throwable` documentation. In addition, we transformed each stack trace for further analysis. We reversed each exception level sequence of call methods and joined the levels at the common methods. Then, the final chain of Listing 1 would be `.main.a.b.c`.

Listing 1 Throwable stack trace.

```
HighLevelException: MidLevelException:
        at Junk.a(Junk.java:13)
        at Junk.main(Junk.java:4)
Caused by: MidLevelException:
        at Junk.c(Junk.java:23)
        at Junk.b(Junk.java:17)
        at Junk.a(Junk.java:11)
        ... 1 more
```

2.3 Identification of Risky API calls

Isolating calls to arbitrary APIs within stack traces of unknown application code called in diverse ways from a larger framework is not trivial. In general, a stack trace of method calls from the Android framework F leading to an exception E, possibly through an application A and an API I. This can be expressed by the following regular expression.

$$((F + (A + I*)*)|(F * (A + I*)+))E$$

This expression reflects several cases in which an exception can occur. For instance consider:

- Within the Android framework: $F + E$
- Within the application: $F * A + E$
- When the application calls an API: $F * A + I + E$
- Within an API-registered application callback: $F * (A + I + A+) + E$
- When an API-registered application callback calls an API: $F * (A + I+) + E$

To locate the API calls that lead to application crashes we had to locate the last instance of an AI pair. We had however no *a priori* knowledge of the methods that belong to the sets F, A, and I. Thus, we used heuristics to determine them. In particular, we constructed from the stack traces n-tuples of length 1–15 anchored to the left hand side of the stack trace and determined their times of occurrence. Looking at the most common ones, we manually established the name space of the Android framework's methods. The sixth most common n-tuple we found was the following:

[1] http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

```
dalvik.system.NativeStart.main
com.android.internal.os.ZygoteInit.main
com.android.internal.os.ZygoteInit$\
  MethodAndArgsCaller.run
java.lang.reflect.Method.invoke
java.lang.reflect.Method.invokeNative
android.app.ActivityThread.main
```

From this 6-tuple we deduced that the framework calls applications through methods that belong to the packages `dalvik.*`, `com.android.*`, `java.*`, and `android.*`. In addition, we searched for other common n-tuples from third parties to fill the framework's name space. For instance, these are the top ones: `com.badlogic.gdx.backends.android.*` and `org.cocos2d.*`.

Knowing the application's name space, we searched the stack traces *backwards* (from the RHS to the LHS) to locate the first place where an application's method called a method that did not belong to its name space (an *AI* sequence). This was, by definition, a call to an API method. In Listing 2, the interesting API call is that to the `setContentView` method.

Listing 2 Exceptional sequence.

```
com.example.Serialize$Looper.run
android.os.Looper.loop
android.os.Handler.dispatchMessage
com.example.SerializeHandler.onMessage
com.example.app.Activity$1.work
android.app.Activity.setContentView
```

Finally, from the stack traces we extracted for further analysis signatures representing the API method (e.g. `android.app.Activity.setContentView`), the exception reported by the API method (e.g. `android.view.inflateException`), and the root exception that triggered the application crash—the exception at the bottom of the stack (e.g. `java.lang.-NullPointerException`). Each signature represents a way in which an API call can fail. Thus, one signature can be associated with many different stack traces and reflects the main cause of a crash. We used the signatures in order to group our data and as a guide for studying the reason of an application failure behind the thrown exceptions.

3 Crash Categories and API Recommendations

Further analyzing our data set, we wanted to see *why* application crashes occur and *what* API deficiencies are responsible for them. To achieve this, we examined the signatures we extracted from the stack traces and we identified major classes of crash causes. In particular, we sorted the signatures according to their number of occurrence and we got the top 600 ones (80% of the stack traces) for analysis. For each signature, we identified the reason of its application failure, and allocated the signature to a broad crash cause category (Figure 1). For the signatures with specific exceptions (e.g. `OutOfMemoryError` and `OutOfBoundsException`) it was easy for us to understand the problems. However, for these with generic unchecked exceptions (e.g. `RuntimeException`, `NullPointerException` and `IllegalArgumentException`) the reason of the execution failure was not clear. Thus, we needed to search in the Android API reference and consulting sites (*stackoverflow*)[2] to

[2] `http://stackoverflow.com`. All sites were accessed on the 20th of July, 2013.

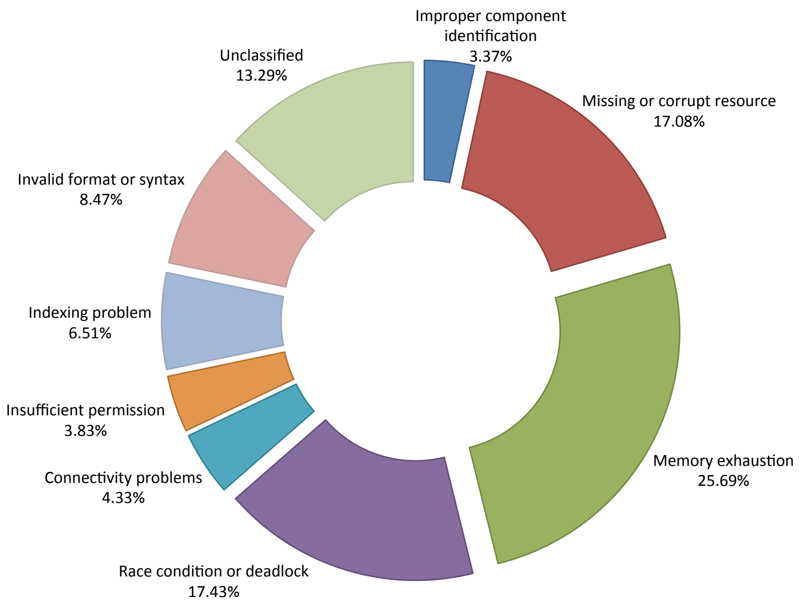

Figure 1 Causes of API-related crashes from top 600 stack traces (80% of total crashes).

reveal the real crash causes. Table 1 presents each crash cause category giving examples of signatures allocated to them and illustrates related API recommendations. Following we discuss each crash category and provide indicative API design and implementation choices.

Memory exhaustion is the most common application crash cause. This was a result we expected, as mobile devices can have constrained memory and developers are seldom aware of the amount of the available memory. Table 1 shows a characteristic example of this category related to a failed import operation for a bitmap. In order to decrease the number of this category's crashes, the API can include an interface for the adaptation of memory consuming resources, so that they can fit in the memory. For instance, if an image cannot be loaded, the developer could sample it first. Moreover, the API can restrict the use of cache structures that trigger memory leaks. Finally, the API can permit the use of file formats that can consume less memory (vector graphics).

Race condition or deadlock is another significant cause of application crashes. This category contains signatures related to: a. database deadlocks, b. race conditions in asynchronous tasks (see Table 1), c. abnormal execution of the lifecycle of an activity, and d. synchronization issues with iterators. To eliminate these crashes, the API should provide non-blocking primitives. Also, developers can catch these problems by using profiling (Traceview[3] and Jinsight[4]) and testing tools [1].

Missing or corrupt resource cause refers, also, to a great number of crashes. In this category, we have added signatures that imply the absence of a resource or inability of the system to decode a resource (see Table 1). We refer to external resources, such as an image or an audio file, and not application components (activities, services, broadcast receivers, and content providers). Crashes because of missing resources can be avoided if the API includes default resources (e.g. layouts). In addition, we found that some exceptions related to this category are unclear (`NullPointerException`). Thus, it is not easy for the developers

[3] `http://developer.android.com/tools/debugging/debugging-tracing.html`
[4] `http://www-03.ibm.com/systems/z/os/zos/features/unix/tools/jinsightlive.html`

Table 1 Categories and Recommendations.

Categories	Signatures	Recommendations
Memory Exhaustion	`android.app.Activity.setContentView` `android.view.InflateException` `java.lang.OutOfMemoryError`	Resource auto-resize interface; Restricted use of cache structures (e.g. LruCache); Use of cheap file formats (vector graphics)
Race Condition or Deadlock	`android.os.AsyncTask.execute` `java.util.concurrent.RejectedExecutionException` `java.util.concurrent.RejectedExecutionException`	Non-blocking algorithms; Specific exceptions
Missing or Corrupt Resource	`android.app.Activity.setContentView` `android.view.InflateException` `java.io.FileNotFoundException`	Default resources; Specific exceptions
Improper Component Identification	`android.app.Activity.startActivity` `android.content.ActivityNotFoundException` `android.content.ActivityNotFoundException`	Useful IDs; Type Checking
Insufficient Permission	`android.app.Activity.startActivity` `java.lang.SecurityException` `java.lang.SecurityException`	Clear documentation; Specific exceptions
Invalid Format or Syntax	`android.database.sqlite.SQLiteDatabase.execSQL` `android.database.sqlite.SQLiteException` `android.database.sqlite.SQLiteException`	Interface for queries on collections (e.g. JQL)
Indexing Problem	`java.util.ArrayList.get` `java.lang.IndexOutOfBoundsException` `java.lang.IndexOutOfBoundsException`	Error-free arguments (iterators); Error ignorance (in loop conditions)
Connectivity Problems	`org.apache.http.impl.client.AbstractHttpClient.execute` `org.apache.http.NoHttpResponseException` `org.apache.http.NoHttpResponseException`	User menu (1. wait, 2. new provider, 3. pause, 4. terminate)
Unclassified	`android.hardware.Camera.open` `java.lang.RuntimeException` `java.lang.RuntimeException`	Clear documentation; Specific exceptions

to understand where a crash comes from. This means that the API should offer specific exceptions regarding problematic resources.

Improper component identification category includes signatures that indicate crashes due to either undeclared components or system's failure to locate a suitable component for a specific task. Crashes of this category can occur because of wrong declaration of the application components (activity, service, broadcast receiver, and content provider). To prevent such crashes, the API can use more meaningful component codes (easy remembered) and appropriate type checks.

Insufficient permission category covers signatures related to crashes because of missing or incorrect activity permissions. Table 1 shows a representative example. The activity cannot start, as the Intent object, which should be passed to the system, has not got the right permissions (for another device to be eligible to receive a message). Specific exceptions and clear documentation provided by the API can eliminate such problems. Static checking tools, also, can help in the early location of permission issues [3].

Invalid format or syntax category refers to crashes due to erroneous method inputs. Specifically, the signatures that belong here imply format problems and invalid syntax of SQL queries. For instance, the corresponding exception in Table 1 reflects that the signature is related to a wrong SQL query syntax. In order to avoid such problems, the API can include an interface for queries on collections (e.g. JQL)[5]. Static checking tools can, also, eliminate

[5] `http://homepages.ecs.vuw.ac.nz/~djp/jql/`

these crashes (consider the *lint*[6] tool).

An **Indexing problem** can be caused by invalid loop conditions and inappropriate structures (see Table 1). Crashes due to these problems can be avoided with the use of error-free arguments (e.g. implicit loops and integer indices), as well as error ignorance (in case a threshold is greater than the size of a list). Also, static checking can solve such issues (consider FindBugs [5]).

Connectivity problems cover signatures associated with networking exceptions (see Table 1). To prevent such cases, the system instead of throwing exceptions can provide the user with a user menu for next actions, such as: 1) wait, 2) choose a new network provider, 3) pause the application, 4) terminate the application. Then, the user has to choose one of these options, and the system can proceed accordingly. Other possible solutions include stress tests and notifications from the system (via monitoring of the network activity).

Unclassified signatures do not give clear information about the real causes of their crashes. For instance, consider a representative example in Table 1 that reflects a crash where the camera cannot be opened. This occurs either because another application is using the camera or because the application has not got the permission to use the camera. However, the `RuntimeException` is generic for one to understand whether the crash cause is a race condition or an insufficient permission. We argue that such exceptions consist an API design problem, and there is a demand for more specific exceptions.

4 Threats to validity

In this section, we discuss the limitations of our study. Internal validity refers to the implications of the method used for the analysis of the stack traces. While, external validity aims to ensure that the findings of our empirical study can be generalized for other samples, too.

4.1 Internal validity

As we had no *a priori* knowledge of the methods that belong to the Android framework, we used heuristics to determine them. We sorted the n-tuples, by their frequency, and looked at the six most common ones to manually establish the name space of the Android framework's methods. In addition, we manually examined other common n-tuples to find application-specific methods. Therefore, although we identified the Android framework's methods that are used to call applications, we may have missed the less common ones, especially from third-party applications.

4.2 External validity

We argue that our findings can be representative of a large population for a number of reasons. First, we believe that for another Android sample we will possibly get the same results. To support this, we examined 600 signatures (rest 20% of the stack traces) through a random sample from our data set, and we validated our crash cause categories against our original data set. Second, we argue that our categories could be the same for another platform because of: 1) the amount of the crashes and applications we examined, 2) the fact that Android runs on a diversity of devices, 3) the size of Android's API, and 4) the generic nature of our categories. We have, however, to validate our results by examining data from platforms, such as iOS and Windows mobile.

[6] `http://developer.android.com/tools/help/lint.html`

5 Conclusions and Future Work

In this work, we presented an analysis of Java stack traces from Android application crashes and an investigation of the causes behind execution failures. As future work, we aim to analyze crash reports from other operating systems such as the iOS and Windows for mobile devices. In addition, we aim to combine the crash reports with other metadata related to specific devices and demographic data. Finally, we work toward to the examination of more stack traces from our sample and the validation of our categories, as well as the automation of our classification method.

Acknowledgements We would like to thank the BugSense Inc., and more specifically its founders Panos Papadopoulos and John Vlachogiannis. Also, we want to thank Panos Louridas, Georgios Gousios, Marios Fragkoulis, and Vassilios Karakoidas for their internal reviews.

This research has been co-financed by the European Union (European Social Fund — ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: Thalis — Athens University of Economics and Business — Software Engineering Research Platform.

References

1 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*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

2 Joshua Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 506–507, New York, NY, USA, 2006. ACM.

3 Patrick P.F. Chan, Lucas C.K. Hui, and S. M. Yiu. DroidChecker: analyzing Android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 125–136, New York, NY, USA, 2012. ACM.

4 Michi Henning. API design matters. *Commun. ACM*, 52(5):46–56, May 2009.

5 David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

6 Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

7 Steve McConnell. *Code Complete, Second Edition*, pages 133–143. Microsoft Press, Redmond, WA, USA, 2004.

8 Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

9 J. Stylos and Carnegie Mellon University. *Making APIs More Usable with Improved API Designs, Documentation and Tools*. Carnegie Mellon University, 2009.

10 Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, pages 1–41, 2013.