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

Charting the API Minefield Using Software Telemetry Data

**Maria Kechagia, Dimitris Mitropoulos ·
Diomidis Spinellis**

the date of receipt and acceptance should be inserted later

Abstract Programs draw significant parts of their functionality through the use of Application Programming Interfaces (APIs). Apart from the way developers incorporate APIs in their software, the stability of these programs depends on the design and implementation of the APIs. In this work, we report how we used software telemetry data to analyze the causes of API failures in Android applications. Specifically, 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 the API calls. We examined a set of more than a half million stack traces associated with risky API calls 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. Given the classes of the crash causes we identified, we recommend API design and implementation choices, such as specific exceptions, default resources, and non-blocking algorithms, that can eliminate common failures. In addition, we argue that development tools like memory analyzers, thread debuggers, and static analyzers can prevent crashes through early code testing and analysis. Finally, some execution platform and framework designs for process, memory, and security management can also eliminate some application crashes.

Keywords Application Programming Interfaces · Stack Traces · Reliability · Mobile Applications

1 Introduction

Application programming interfaces (APIs) are the unsung heroes of information technology. Unlike algorithms, databases, programming languages, and operating systems, there are no courses or academic chairs dedicated to them. Yet, all these

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

disciplines would be useless without suitable APIs. More importantly, any non-trivial modern program depends on usable, robust, and efficient APIs to implement a significant part of its functionality.

Recent work has evaluated API usability and documentation (Stylos and Myers 2008; Stylos 2009; Robillard 2009; Farooq et al 2010; Robillard and DeLine 2011; Maalej and Robillard 2013). There is however scant empirical evidence regarding API design and implementation guidelines; published articles focus on general practices (Bloch 2006; Henning 2009; Tulach 2012). In addition, although there is a body of research on bug characterization (Li et al 2006; Guo et al 2010; Tan et al 2013) and crash analysis (Ganapathi et al 2006; Ganapathi and Patterson 2005; Kim et al 2011a), to the best of our knowledge, there is no study that attributes crash causes to API deficiencies. The main two contributions of this paper are: a) a method that links telemetry data from application crashes to API calls, and b) the use of this method to identify API weaknesses that can lead to execution failures.

In this context, we aim: a) to show API design and implementation deficiencies that commonly lead to application crashes in mobile software and b) to improve the design and use of mobile software APIs, based on clear documentation, appropriate debugging tools, as well as frameworks and platforms for crash prevention.

To meet our goals, we conducted an empirical study on software telemetry data from mobile application crashes. Specifically, we processed that data to extract more than a million stack traces, stitching together parts reported separately as elements of a chained exception. We then established heuristic rules to draw the border between applications and API calls. This allowed us to pin down crashes to specific API calls, and analyze the causes behind the most common problems. Our results show that the top crash categories can be attributed to memory exhaustion, race conditions or deadlocks, and missing resources. We were however unable to classify the crash causes for a significant number (almost 10%) of signatures associated with generic exceptions (`RuntimeException`, `NullPointerException`), which possibly stem from programming flaws and poor API documentation.

Then, we conducted a qualitative study and a short bibliographic survey to find indicative solutions that could diminish crashes similar to those of our sample, and improve mobile applications' user experience. In particular, we examined a set of representative crashes from each crash cause category and we concluded on API recommendations for **fault tolerance** in mobile software. Also, we found in bibliography tools for **fault localization** that can aid developers to write applications with fewer faults and execution frameworks and platforms that can offer **fault prevention** to common mobile software problems (process and memory management, and connectivity issues). For the understanding of software faults and associated fault elimination methods consider the work of Avizienis et al (2004).

We chose to focus our study on the APIs rather than other sources of crashes for a number of reasons. First, to address an **imbalance of information**. Whereas application and operating system builders can obtain a wealth of information regarding the design and implementation of their code through the results of testing and, increasingly, telemetry, these results are unlikely to reach API designers, who work or are perceived to work in a different area. (An exception to this observation may be vertically-integrated companies, such as Microsoft and Oracle, which develop code along the whole system stack.) Reports on crashes that could have been avoided through a better API design and implementation, land on the hands of application builders, who address them by fixing their applications on a case by case basis. This brings us to the second reason, namely **impact**. Locating deficient APIs and improving their design or implementation can improve the stability of thousands of applications that use them. Then comes an unashamedly **practical aspect**. Performing a wide-range analysis of application software errors based on telemetry data would be difficult, because the code of applications submitting such data is typically not openly available. In contrast, the code of most APIs we encountered in our study is available as open source software. Finally, we subscribe to the view that there are many **interesting research** topics associated with the design and implementation of APIs, such as: 1) prediction of application crashes depending on API version changes, 2) fault prevention based on API design choices (error handling), and 3) guidelines and quality metrics for APIs of different domains.

The contributions of this work are:

– a method for stratifying elements of arbitrary crash stack traces into the operating system framework, the application, and the API,
– a presentation of APIs that, according to empirical data, lead to application crashes,
– an analysis of API design and implementation choices that are often the root cause of application crashes, and
– an overview of indicative API recommendations, tools, and execution platforms and frameworks able to decrease the number of application crashes.

Although this study was conducted by using data from a single system—Android, this platform provides a valuable subject for a broad study regarding API evaluation. First, the Android API is quite large and lucrative for examination.[1] Second, the implementation of its API is open source, and, thus, we can get and study its code. Finally, Android is a leading platform, which runs on millions of devices, and its API is approximately used by 700,000 applications. This is important for the assessment of an API, as we can observe its use on samples of a lot of diversity.

In the rest of this paper, we first outline related work (Section 2). We then describe the methods of our study (Section 3). In Section 4 we discuss descriptive results and the crash cause categories we found. In Section 5 we present a summary of API recommendations, tools, and execution platforms and frameworks able to reduce the number of the crashes that belong to the crash cause categories. We end up with our threats to validity in Section 6 and conclude with an overview of the big picture and directions for future work in Section 7. Finally, Table 1 contains terms we broadly use in the next sections.

2 Related Work

Our approach is related to previous work in the following areas, namely: software telemetry, crash cause analysis, API evaluation, and reliability analysis in Android.

2.1 Software Telemetry

Telemetry is a technology that allows data measurements and analysis to be made from a distance. In software engineering, telemetry has been used in several manners.

[1] `http://developer.android.com/reference/packages.html`

Table 1: Terms and definitions.

Term	Definition
Bug	An error or a programming defect that can lead to a crash.
Crash	Unexpected termination of a program.
Crash report	Detailed record of a crash that is comprised of a stack trace and other runtime metadata.
Stack trace	Ordered chain of frames (each associated with a method signature).
Signature	Part of a stack trace directly associated with the crash cause (Section 3.3).
Crash cause	The reason of an application execution failure, highlighted by the stack trace.
Crash category	A group of similar crash causes (e.g. memory exhaustion or race conditions).
Exception	"An event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions."[a]
Checked exception	"Exceptional conditions that a well-written application should anticipate and recover from."[b]
Unchecked exception	Exceptional conditions that are external (`Error`) or internal (`RuntimeException`) to the application, and that the application usually cannot anticipate or recover from.[c]

[a] `http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`
[b] `http://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html`. See checked exceptions.
[c] `http://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html`. See unchecked exceptions.

A notable case involves the collection of project-related data via sensors attached to software development tools to support project management decision making (Johnson et al 2005). The sensors send project-related data back to a server, and project members can access the data for analysis via the web. Telemetry has also been employed to ensure software dependability (Gross et al 2006). This approach involves the instrumentation of a system for the continuous monitoring of its runtime state. The collected data can be used for the prediction of system failures using statistical models. A similar method has been used for the detection of software aging signs in complex systems (Gross et al 2002). In this study, we used software telemetry data from a centralized crash report management service to identify API deficiencies. The data—stack traces—stem from Android application failures.

2.2 Crash Cause Analysis

Crash reports from operating systems and applications can give researchers valuable insight into the reasons behind execution failures. This is important for the development of effective debugging and testing tools, and consequently for software quality, reliability, and dependability (Endres 1975; Lee and Iyer 1995; Sullivan and Chillarege 1991; Ganapathi et al 2006; Tan et al 2013). In this study we examined crashes in mobile software, which involves particular challenges: restricted memory, multithreading applications, connectivity issues, and user permissions.

2.2.1 Categorization Approaches

The categorization of execution failures can contribute to the understanding and organization of software faults and execution failures. Ploski et al (2007) provided a comparison among different categorization approaches of software faults. They studied six distinguished works (Knuth 1989; Beizer 2003; Gray 1986; Chillarege et al 1992; Eisenstadt 1997; DeMillo and Mathur 1995) and presented a roadmap for the development of a fault categorization schema. We took into consideration this roadmap for the organization of our study. In the following paragraphs, we present three empirical studies that are close to our approach and compare them with it.

Sullivan and Chillarege (1991) classified reports from software defects for a high-end operating product of IBM to highlight common software defects. Specifically, they identified two defect classes: overlay errors due to memory errors and regular errors due to data, synchronization, compilation, and logic errors. Their findings show that most crashes come from memory and synchronization causes. Fifteen years later, we found that the same causes are dominant in mobile software. This fact possibly spells a need for more effective tools for fault tolerance and better support from system designers to developers.

Recently, Tan et al (2013) reported bug characteristics from Apache, Mozilla, and the Linux kernel and investigated bugs that can evolve in crashes. Even though we focused on the stack trace domain, rather than the temporal domain looking for API design and implementation deficiencies in mobile software, our findings agree in many cases. We too found that memory exhaustion is a major crash cause (23%) in mobile software. One possible reason for this can be the restricted memory of the mobile devices and devices' sensibility to respond to orientation changes. In addition, we identified that hangs and crashes due to concurrency issues form another significant category that matches the findings of Tan et al (2013) for the Linux kernel. Thus, we agree with Tan et al (2013) that both memory and concurrency issues are the most severe causes of crashes and hangs in mobile software (40%). Finally, we similarly found that a significant number (almost 20%) of crash causes (indexing problem and invalid format or syntax) could be associated with semantic bugs.

Crash causes can be classified based on the manifestation of crashes in different system components aiming to improve software reliability and dependability. Characteristically, Ganapathi et al (2006) analyzed crash reports, from the Windows XP kernel, to identify relationships between crashes and failure causes and improve operating systems' dependability. Their findings show that most crashes occurred in the operating system core, the graphics and application drivers. In our study, we also classified crash data but based on API crash causes and not on the location (including hardware) of the manifested crashes.

2.2.2 Impact

Root cause analysis can be used in fault tolerance and prediction, contributing in the improvement of debugging efforts. For instance Kim et al (2011a) analyzed crash reports from Firefox and Thunderbird and applied machine learning techniques to predict the top crashes to be scheduled for early fixing. In addition, Podgurski et al (2003) applied feature selection, clustering, and visualization to group system failures and aid programmers to prioritize their debugging efforts. Finally, Shelton et al (2000) conducted robustness testing for the Microsoft and Linux APIs to compare the capabilities of both operating systems. Here, we analyzed crash data to examine common crash causes of mobile software and make possible fault tolerance suggestions to overcome API deficiencies.

2.2.3 Mining Techniques

In addition to the manually and machine learning categorization of the crash reports, stack traces can be grouped using "bucketing" and execution path reconstruction. In particular, Dang et al (2012) presented a technique based on call stack matching. This technique measures the similarities of call stacks and assigns the reports to appropriate categories ("buckets"). Kim et al (2011c) proposed an approach based on crash graphs, which are an aggregated view of crashes. Then, they used the graphs to improve categorization by winnowing out crashes between similar graphs. Liblit and Aiken (2002) studied the reconstruction of execution paths based on partial execution information like backtracks. Their technique can assist developers to find the root cause of the crash. In our study, we used elements from both methods ("bucketing" and execution paths) to find similarities in method calls from the stack traces and visualizing the execution paths as graphs (see Figure 2).

2.3 API Evaluation

An important topic in software engineering is the development of APIs that can improve developers' productivity and resulting products' quality. Many papers study the assessment of APIs based on usability tests and the evaluation of their documentation. Additionally, many specialized tools have been developed to support the study of API usability and learnability. Recently, Robillard et al (2013) published a detailed survey on such tools and mining techniques regarding API analysis.

2.3.1 Usability

Usability is a well-documented topic regarding APIs. Clarke (2004) presented a framework of cognitive dimensions for the assessment of API design decisions in terms of usability. A stream of comparative studies has dealt with the usability outcomes of particular API design decisions and their impact on developers' productivity. Specifically, Ellis et al (2007) argued that the factory pattern is more efficient than constructors, while Stylos and Clarke (2007) found that developers are more productive when using APIs that do not require constructor parameters. The efficient use of APIs has been also examined by Kawrykow and Robillard (2009) through source code and byte code analysis. In particular, they checked whether developers write code that replicates the behavior of a library method without actually calling it. Looking at the choice of an API from the developers' side, de Souza and Bentolila (2009) presented an approach that allows developers to evaluate API usability based on complexity metrics. Finally, learning theories have also been used in the usability evaluation of APIs (Gerken et al 2011). Taking into account API usability problems reported in the above studies, we tried to identify application crash causes that could be related to poor API usability. We also searched for crashes related to API documentation quality and learnability.

2.3.2 Documentation Learnability

Documentation quality is directly associated with the developers' productivity and the effort required for software maintenance. There is a growing research interest for the identification of the obstacles that developers face when they use an API's documentation. For instance, Robillard and DeLine (2011) introduced API documentation approaches, such as code examples, API use scenarios, formatting, and presentation, that can increase productivity and overcome learning barriers

Recently, Maalej and Robillard (2013) identified patterns of knowledge in API reference documentation and grouped them into a taxonomy that can be used for API evaluation and content organization. In addition, Buse and Weimer (2012) developed an algorithm that synthesizes human-readable API usage examples that can assist developers when they study the API documentation. Finally, Shi et al (2011) conducted a quantitative study on the evolution of API documentation of five real-world Java libraries. In our study, we looked up the methods participating in the stack traces in the Android API reference and categorized the thrown exceptions according to their recorded causes. Our empirical evidence suggests that many crashes are associated with unclear documentation (see Section 4.2.8).

Finally, Sproull and Waldo (2014) first argued that performance issues related to API calls should be documented in the API reference to meet *contract* expectations between the caller and the implementation. In our study, we empirically showed that there were many cases where the API documentation provided no hints about the performance of the API methods. However, since mobile devices have significantly restricted resources, it seems to be quite important for the developers to know from the API documentation the resources particular methods are consuming. For instance, we found that the top crash cause of our sample can be attributed to memory exhaustion (23%). However, there was no information about methods susceptible to memory exhaustion and related exceptions in the API reference.

2.4 Reliability Analysis in Android

On the recent years, the market of mobile software has boomed. Thus, a strand of research has focused on the investigation of mobile software platforms and applications. As Android is an open source platform, the majority of the studies on mobile software analyzes Android's operating system and its applications' source code. Most works examine security issues regarding application permissions and access control policies (Shabtai et al 2012, 2010; Enck et al 2009; Ongtang et al 2010), as well as performance and energy management concerns (Kim et al 2012; Vallina-Rodriguez and Crowcroft 2013). Given however that millions of developers use the Android API reference and its source code to implement their ideas, there is a demand for a rigorous investigation of the Android API.

Felt et al (2011) analyzed the Android source code to pinpoint API permission leaks. They found that a lot of Android applications are overprivileged because developers fail to understand the privileges referred in the Android API documentation. In addition, Linares-Vásquez et al (2013) examined the implementation of different Android API versions and argued that heavy bug fixes and changes in the API have negative impact on the ratings of Android third-party applications. Maji et al (2010) also explored two mobile operating systems, Android and Symbian, and grouped their bugs based on bug reports from issue tracking systems. Their categories were associated with the system segments where the bugs were manifested. Then, they categorized appropriate source code bug fixes. In this study, we judge the Android API from many different angles (design and implementation), based on the causes of the manifested application crashes. We add to the previously discussed works by pinpointing insufficient documentation about method exceptions and missing fault recovery mechanisms of the provided interfaces.

3 Methods

To understand API deficiencies based on the analysis of stack traces and make relevant suggestion for crash prevention, we followed the steps below.

1. We cleaned and processed a set of almost one million stack traces. From each stack trace, we kept a characteristic part—**signature** (see Table 1 and Section 3.3). A signature succinctly represents the salient API related data associated with the crash.
2. We sorted the signatures based on their number of occurrences in the stack traces and we split them in three subsets, based on the packages of the included methods—`(com.)android.*`, `java.*`, and `com.*`. We examined the crash causes for each subset and we organized our findings into eight broad categories.
3. Having mapped the space of the most common causes of application crashes, we made indicative API design and implementation recommendations. Specifically, we drew and examined a set of representative signatures from each crash cause category and we sought possible suggestions that could eliminate such crashes. We also conducted a survey of development tools and execution platforms and frameworks able to prevent execution failures.

3.1 Data Provenance and Description

The subject of this study concerns stack traces from Android mobile application crashes that were collected through a centralized crash report management service.

Android is an embedded device based on the Linux operating system (kernel version 2.6) capable of hosting mobile applications. Figure 1 illustrates the architecture of the Android platform running on a mobile device. The Linux kernel is at the bottom layer and forms the border between the hardware and the rest of the software. It offers services such as memory management, process management, networking, power management, and drivers (flash memory, bluetooth, Wi-Fi, keyboard, audio). Dalvik is the virtual machine in the middle layer, which is essential for running different applications at the same time. Each application runs as a separate process, having its own virtual machine instance. Android, also, includes C/C++ libraries used by a lot of components (e.g. SQLite, SSL, Media Framework, Surface Manager) through the Java Native Interface (JNI). Finally, the Android platform hosts several standard applications (e.g. contacts, browser, phone) and others provided by third parties. All the Android applications are written in Java and use APIs from the Android framework, third-party libraries, and the Java Software Environment (JSE).

Every Android **application** can contain four component types: *activities* associated with the user interface screens, *services* that run in the background (e.g. playing

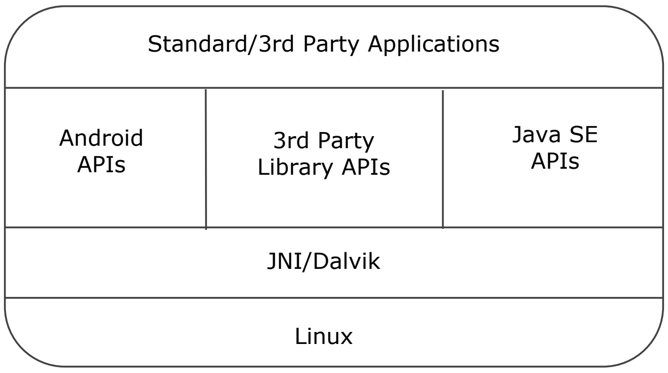

Fig. 1: The Android Platform Architecture.

music or fetching data), *content providers* that manage shared application data (either in an SQLite database or on the web) and *broadcast receivers* that respond to broadcast announcements (i.e. when the data downloading has been completed). These components can be activated through messages called `Intent` objects. `Intent` objects carry a description of actions to be accomplished by a component, so that the system can select the right component for their execution. In addition, Android uses an `AndroidManifest.xml` file per application to document each application's components. This file contains not only information about the components themselves, but also about other application requirements such as the version of the operating system, Java packages, and libraries' names. Finally, this file defines the permissions the application requires to interact with other applications or external sources.

The provider of our data set, **BugSense Inc.**,[2] is a privately held company founded in 2011, and based in San Francisco. Its aim is to provide error reporting, analytics, and insights regarding the performance and quality of mobile applications. Currently, about 2535 iPhone and 4755 Android applications send stack traces to BugSense. An application can send reports to BugSense by installing an SDK and adding one line to its source code. If the application crashes, the stack trace is sent to BugSense's servers in JSON format, and then stored in the Google App Engine Data store[3] and a separate database. Application vendors can check this data by logging to BugSense's dashboard.

We chose to analyze a dataset of crash reports from Android applications for the following reasons: 1) the vast number of stack traces from mobile applications, 2) the Java language Android applications are based on, and 3) the ease with which the collected data could be manipulated. Our sample comes from 1,629,940 stack traces collected in real time from the January 13th to April 11th, 2012. The data set was extracted between the April 15th and May 1st, 2012, by running a script based on

[2] `https://www.bugsense.com/`. Recently acquired by Splunk Inc.

[3] `https://developers.google.com/appengine/docs/python/datastore/`

the Google Data store Python API. The stack traces come from 4,618 applications and the Android API refers to versions from 1.0.0 to 4.1.1.

3.2 Data Cleaning

We started the data analysis by cleaning our data set. Executing SQL queries, we isolated from BugSense's dump of the crash reports 920,437 records associated with Android applications. Each entry comprised a key, an error identifier, an application key, and a stack trace. For the purposes of this study, we kept only the stack trace.

As the format diversity among the stack traces was significant, we selected the stack traces or parts of the stack traces that were in line with the documented format of the `printStackTrace()` method, from the `Throwable`[4] Java class. For this, we wrote a parser in Python,[5] using regular expressions and dictionaries as data structures. Algorithm 1 describes in pseudocode the basic routine of our Python script, which we used to isolate the well-formed stack traces for our analysis.

This involved removing 2.08% of the initial stack traces. Thus, after parsing, we kept 901,274 well-formed stack traces. Listing 1 illustrates a representative chained (having more than one exception level) stack trace from our data set. In accordance with the exceptions' hierarchy in the `printStackTrace()` Java method, the `RuntimeException` in line 1 of Listing 1 refers to the `HighLevelException` and the `IndexOutOfBoundsException` in line 14 refers to the `LowLevelException`. Finally, line 27 represents the "enclosing" exception and it highlights the end point when parsing a stack trace with chains.

Listing 1: A representative stack trace of an application crash

```
1  java.lang.RuntimeException: java.lang.IndexOutOfBoundsException
2  at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:1768)
3  at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:1784)
4  at android.app.ActivityThread.access$1500(ActivityThread.java:123)
5  at android.app.ActivityThread$H.handleMessage(ActivityThread.java:939)
6  at android.os.Handler.dispatchMessage(Handler.java:99)
7  at android.os.Looper.loop(Looper.java:130)
8  at android.app.ActivityThread.main(ActivityThread.java:3835)
9  at java.lang.reflect.Method.invokeNative(Native Method)
10 at java.lang.reflect.Method.invoke(Method.java:507)
11 at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:864)
12 at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:622)
13 at dalvik.system.NativeStart.main(Native Method)
14 Caused by: java.lang.IndexOutOfBoundsException
15 at android.content.res.StringBlock.nativeGetString(Native Method)
16 at android.content.res.StringBlock.get(StringBlock.java:82)
17 at android.content.res.AssetManager.getResourceValue(AssetManager.java:217)
18 at android.content.res.Resources.getValue(Resources.java:925)
19 at android.content.res.Resources.getDrawable(Resources.java:587)
20 at com.android.internal.policy.impl.PhoneWindow.generateLayout(PhoneWindow.java:2321)
21 at com.android.internal.policy.impl.PhoneWindow.installDecor(PhoneWindow.java:2356)
22 at com.android.internal.policy.impl.PhoneWindow.setContentView(PhoneWindow.java:209)
23 at android.app.Activity.setContentView(Activity.java:1657)
24 at com.example.TabsActivity.onCreate(TabsActivity.java:53)
25 at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1047)
26 at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:1722)
27 ... 11 more
```

[4] http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Throwable.html
[5] Our source codes can be found in: https://github.com/mkechagia/stack-traces. For the cleaning see parsing_st.py file

Algorithm 1 Stack Trace Cleaning

```
 1: Input: stack trace in list form
 2: Output: stack trace in dictionary form
 3:
 4: procedure KEEPSTINTODICT(list)
 5:     # initialize the dictionary (key: exception level, [values]: methods in this level)
 6:     dict = OrderedDict([])
 7:     # initialize the exception level
 8:     level = “type”
 9:     for each $i$ in list do
10:         # validate that the method is part of an exception level chain[a]
11:         if (list[$i$] begins with “^$at\s$”) and (isMethodValid() == true) then
12:             # update the level if the current frame belongs to the highest exception level
13:             if (list[$i-1$] is the “HighExceptionLevel”) then
14:                 dict ← dict[high_level]
15:             end if
16:             # add a new method as value in the current exception level
17:             dict[level] ← append(list[$i$])
18:         else if (list[$i$] begins with “Caused\$s$”) and (level ! = “type”) then
19:             # validate the current exception level; check methods and end point
20:             if (isChainWellFormed() == true)[b] then
21:                 # add new exception level in dictionary keys
22:                 dict ← dict[new_level]
23:             end if
24:         else if (list[$i$] begins with “\.\.\.\$s$[\$d$]+ \$s$more”) and (level ! = “type”) then
25:             # remove the last method from the values of the current exception level
26:             dict[level] ← remove(list[$i-1$])
27:         else if (list[$i$] begins with “^\$s*$$”) then
28:             # stop processing; there is an empty line in the stack trace
29:             break
30:         else
31:             continue
32:         end if
33:     end for
34:     # return a dictionary with the current stack trace
35:     return dict
36: end procedure
```

[a] By checking the existence of “Caused\$s$” and/or “\.\.\.\$s$[\$d$]+ \$s$more” in the next frames.
[b] According to the documentation of the `printStackTrace` Java method.

After parsing the stack traces, we kept only the method names, linked chained exceptions, and reversed their order to follow the caller-callee direction. Listing 2 illustrates the parsed chain corresponding to the stack trace of Listing 1. In particular, for each valid stack trace, which was stored in a dictionary (see Algorithm 1), our script reverses the list of the values (frames) for each dictionary key (exception level). To get Listing 2, the script reversed the order of lines 1-13 (`HighLevelException` chain) of Listing 1 and, then, the order of lines 14-26 (`LowLevelException` chain) of Listing 1. Finally, the script removed the line 26 of Listing 1, which is the link between the two exception level chains of our example. See also the relevant line, 26, in Algorithm 1.

Listing 2: Parsed stack trace

```
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
android.os.Looper.loop
android.os.Handler.dispatchMessage
android.app.ActivityThread$H.handleMessage
android.app.ActivityThread.access$1500
android.app.ActivityThread.handleLaunchActivity
android.app.ActivityThread.performLaunchActivity
!java.lang.RuntimeException
android.app.Instrumentation.callActivityOnCreate
com.example.TabsActivity.onCreate
android.app.Activity.setContentView
com.android.internal.policy.impl.PhoneWindow.setContentView
com.android.internal.policy.impl.PhoneWindow.installDecor
com.android.internal.policy.impl.PhoneWindow.generateLayout
android.content.res.Resources.getDrawable
android.content.res.Resources.getValue
android.content.res.AssetManager.getResourceValue
android.content.res.StringBlock.get
android.content.res.StringBlock.nativeGetString
!java.lang.IndexOutOfBoundsException
```

Figure 2 depicts a graph of method call chains, from the most common 15 parsed stack traces, that lead to exceptions. It is evident that a method can be called by many others that exist in different stack traces (incoming edges), and, thus, it can appear several times. This metric is an indicator of the importance of a method, but this is out of the scope of this work. The graph illustrates the complexity of the method calls and the possibility of finding risky API calls from stack traces to locate problems into the API. In the following section we discuss how we located risky API calls from the processed stack traces.

3.3 Locating Risky API Calls

To organize our sample and focus on application crashes due to API defeciencies we used a heuristic method that identifies risky API calls in stack traces. We did this since we had also stack traces with only application calls in our sample and we wanted to extract only the stack traces with API calls.

With the term *risky* API call we refer to a call from a method of an Android application to an API method that is probably responsible for an application crash. The main reasons that a call to an API method could lead a program to an execution failure are: 1) an inappropriate call to an API method from a method of an application (i.e. implementation defect of the application), 2) defect in the implementation of an API method (i.e. indexing problems, cache structures of an arbitrary size that lead to memory leaks, careless use of the `string` type for resource codes), 3) defect in the design of an API method (i.e. missing prevention mechanisms for heavy memory consumption, undocumented exceptions for common runtime errors, unclear documentation about permissions, lack of useful mechanisms for multithreading applications). In the rest section we describe how we locate such calls in stack traces from application crashes and why these calls are risky. In Table 2 we explain the types of the method calls we examine and in Table 3 we provide representative examples.

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, possibly from the Android framework F leading to an exception E, possibly through an application A and an API I, can be described through the

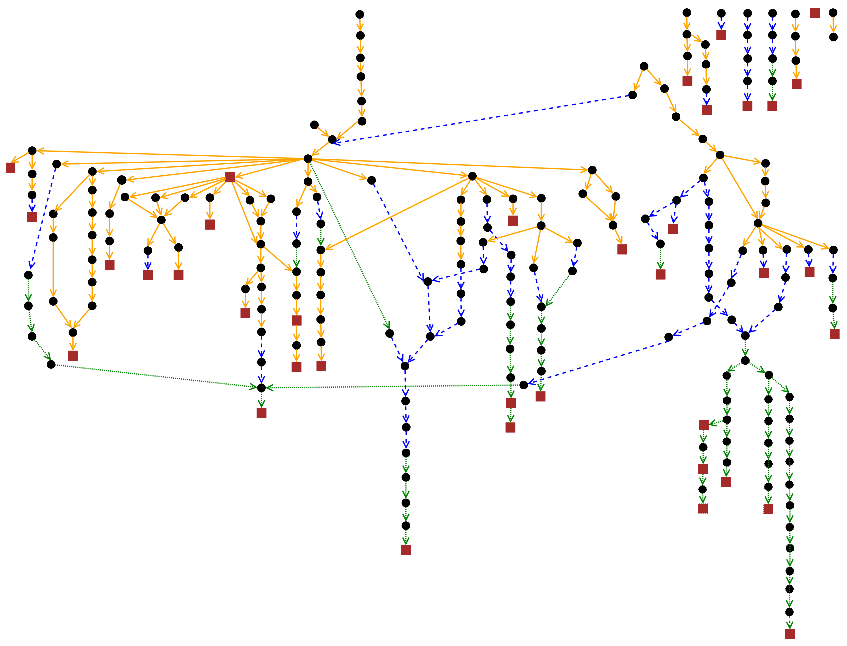

Fig. 2: A graph of the 15 most common stack traces representing 41 thousand cases. Solid line (orange): calls from the Android framework; dashed line (blue): application calls; dotted line (green): API calls; square: exceptions; circle: methods.

Table 2: Call Types and Definitions.

Type	Definition
Framework call	When an Android application is launched the Android framework calls a series of methods before it calls the application, which in turn calls an entry method such as `main()`. We name these calls framework calls.
API call	An API call is made when a method of a client application calls an API method. An API method is a method of a class not belonging to application packages. We examine calls to three API types: 1) Android, 2) Java, and 3) third-party.
Application call	An application call can take place between two methods of a client application or from an API method to a method of the client application.

following regular expression.

$$((F+\ (A+\ I*)*)\mid(F*\ (A+\ I*)+))E$$

This expresses various scenarios in which an exception can occur (see some examples in Figure 2).

Case 1 Within the Android framework: $F+\ E$
Case 2 Within the application: $F*\ A+\ E$

Case 3 When the application calls an API: $F* \; A+ \; I+ \; E$
Case 4 Within an API-registered application callback: $F* \; (A+ \; I+ \; A+)+ \; E$
Case 5 When an API-registered application callback calls an API: $F* \; (A+ \; I+)\{2,\}E$

To locate API calls that lead to application crashes our goal is to locate the last instance of an AI pair (see Section 3.4.1). For the identification of the API methods in the stack traces, we chose not to use the Android reference for two main reasons. First, we would like to make our method generic and applicable to similar systems for stack trace analysis. Second, we would like to examine calls to different types of APIs, from Android, Java, and third-party (see the different stacks in Figure 4 and 5, in Section 4.2). Then, as we had no *a priori* knowledge of the methods that belong to the sets F, A, and I, we used the following process and heuristics to determine them.

First, we deduced the name space of the Android framework's methods. We reasoned that due to the common way in which the Android framework calls the applications, the corresponding method sequences $F+$ would appear considerably more often than application methods $A+$. We therefore constructed from the reversed stack traces n-tuples (see Listing 2) of length 1–15 anchored at the left hand side of the stack trace and determined their frequency. Each element of the n-tuple was the name of a method. By ordering the n-tuples by their frequency and looking at the most common ones we manually established the name space of the Android framework's methods. For instance, Listing 3 illustrates the most common 6-tuple.

Listing 3: The most common 6-tuple

```
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.*`, as well as `android.*`. By examining other common n-tuples we ended up including in the framework name methods from two more package name spaces: `com.badlogic.gdx.backends.android.*` and `org.cocos2d.*`. These are two game engines that appeared to be calling application-specific methods.

Then, by stripping from each stack trace the method calls of the Android application framework ($F*$), we isolated the name space of each application through the first method that appeared in the stripped-down stack trace. We defined as the application's methods (A) those that belonged to the name space defined by the first two dot-separated elements of the package name (e.g. `com.example` or `org.umlgraph`). This allowed for the possibility of an application using diverse name spaces under a given domain for a single application at the expense of considering calls to an API developed by the same entity as part of the application (e.g. a call from `com.example.GameActivity.onCreate` to `com.example.lib.Proxy.init`).

With the knowledge of the application's name space at hand we searched the stack trace *backwards* (from the RHS to the LHS) to determine 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. We searched for the call backwards, to handle cases where the application registers a callback handler that will later call an API method. In such a case the problematic call is not the API call used to register the callback handler method, but the second API call. For instance, in the Listing 4 below, the interesting API call is that to the `setContentView` method, rather than the one to `loop`.

Listing 4: 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 represents directly their crash cause. We used the signatures in order to simplify our data set and as a guide for studying the reason of the application failure behind the thrown exceptions.

We generated the signatures and analyzed the number of their repeated occurrences in two ways by creating two multisets.

1. A multiset (**unique**) where each problem signature—as identified by its stack trace—appears only once. The multiplicity of each signature's occurrence reflects how many *applications* (or distinct parts of an application) were affected by the problem.
2. A multiset (**total**) of all stack traces reported over the analyzed period, without any de-duplication applied to them. The multiplicity of each signature's occurrence reflects how many *users* were affected by the problem and to what extent.

3.4 Validation of the Experimental Method

To show that our method in Section 3.3 locates risky API calls, we validated our algorithm with three experiments based on publicly available software and data.

3.4.1 Identification of risky API methods in real applications

First we showed that the last AI pair in a stack trace represents a risky API call (Section 3.3). To prove this, we downloaded application samples from the Android SDK, built them, and experimented on the included API methods and the produced stack traces. Specifically, we built and run the application sample `ContactManager` and we checked the characteristic Case 3 in Section 3.3.

To prove Case 3 ($F*\ A+\ I+\ E$), we changed the input of the `setContentView` method to `null` (see in the `onCreate` method of the `ContentManager` class.) Table 3

Table 3: An application calls an API and crashes.

F: Framework	dalvik.system.NativeStart.main
	com.android.internal.os.ZygoteInit.main
	com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run
	java.lang.reflect.Method.invoke
	java.lang.reflect.Method.invokeNative
	android.app.ActivityThread.main
	android.os.Looper.loop
	android.os.Handler.dispatchMessage
	android.app.ActivityThread$H.handleMessage
	android.app.ActivityThread.access$600
	android.app.ActivityThread.handleLaunchActivity
	android.app.ActivityThread.performLaunchActivity
	!java.lang.NullPointerException
	android.app.Instrumentation.callActivityOnCreate
	android.app.Activity.performCreate
A: Application	com.example.android.contactmanager.ContactManager.onCreate
I: API	android.app.Activity.setContentView
	com.android.internal.policy.impl.PhoneWindow.setContentView
	com.android.internal.policy.impl.PhoneWindow.setContentView
	android.view.ViewGroup.addView
	android.view.ViewGroup.addView
	android.view.ViewGroup.addViewInner
E: Exception	!java.lang.NullPointerException

shows a `NullPointerException` produced by the `setContentView` method, assuming that the passed `view` was missing for some reason. Then, we applied the cleaning method (Section 3.2) and the method for the location of risky API calls (Section 3.3) to the produced stack trace, and we indeed located the `setContentView` method as the root cause of the crash. Finally, in the Appendix there are examples of our stack traces that match the remaining cases with API calls (Case 4 and Case 5) from the regular expressions in Section 3.3.

3.4.2 Verification of the risky API methods

We also searched in the source code of the Android API for the methods included in our signatures. In particular, we downloaded the version 15 of the Android API (latest version with the most stack traces in our sample), parsed the source code using a simple Java doclet,[6] and we extracted documented methods from the Android API. We took into account the public and protected methods as these methods should be in the API reference. We found that 96% Android and 99% Java methods from our signatures were documented in the Android API reference. This means that our method indeed found a significant number of Android and Java API methods. However, we were not able to check all the third-party libraries associated with the Android framework, except for specific cases such as: `com.badlogic.gdx.-backends.android.*`, `org.cocos2d.*` and `com.google.android.*`. We discuss this limitation in Section 6.1.

[6] http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/doclet/overview.html

3.4.3 Location of the extracted signatures in public sources

Finally, we validated our findings by showing that the risky API methods we found seem to confuse Android developers. For this, we used challenge data from *stackoverflow.com*[7] and Android issue tracker.[8] We downloaded these data sets from the site of the Mining Software Repositories (MSR) workshop (Bacchelli 2013; Shihab et al 2012). Then, we got 6,199 titles of discussion threads from *stackoverflow.com* and 20,169 titles from the Android bugs to search for pairs of Android API methods and exceptions. The location of the Android discussion threads in the *stackoverflow.com* was trivial, because of the provided Android tags. In addition, we easily extracted the titles from all threads as both data sets were in `xml` format. Then, we isolated from the titles referred methods and exceptions and we looked for them in our signatures. To extract the methods and the exceptions from the titles we used the following regular expressions, respectively:

$$[a-z]+\,[A-Z][a-z]+[A-Za-z]*$$

$$[a-zA-Z]+\,(Exception \mid Error)$$

In other to be sure that the methods and the exceptions we identified in our challenge data exist in the documentation of the Android API, we used the Java doclet and the documented methods from Section 3.4.2. We found 180 distinct *stackoverfow.com* methods that exist in the documentation of the Android API reference. From these methods 133 were in our signatures (74%). In addition, we found 227 distinct methods from the Android bugs that are documented in the Android API reference and 122 of these that exist in our signatures (54%). Finally, we checked the pairs of methods and exceptions of both data sets and we compared them with those in our signatures. We found 199 distinct pairs with documented Android methods in *stackoverflow.com* from which 131 were in our signatures. Also, we identified 21 distinct pairs of methods and exceptions in Android bugs existed also in the Android API reference and 8 of these were in our signatures.

3.4.4 Conclusions

In this section, we presented three experiments we conducted for the validation of our experimental method by using publicly available software and data. First, we explained why the last *AI* pair in a stack trace can be associated with a risky API call. Second, we showed that more than 90% of the methods of our stack traces where also in the source code of the Android API (including Java API methods). Finally, we analyzed two data sets of challenge data and we found that 74% methods of our signatures also exist in *stackoverflow.com* and 54% in Android bugs. In the next sections, we present the results of the analysis of our stack traces that stem from a large data set of Android application crashes.

Table 4: Key metrics of the analyzed data.

Elements	Total	Unique
Stack traces	901,274	101,279
Method calls	16,697,379	193,064
Class names in an Android context	10,693,844 56.5%	2,207 3.1%
Class names in an app context	2,712,754 14.3%	61,379 85.5%
Class names in an API context	4,192,055 22.1%	10,349 14.4%
Class names total	18,928,752	71,771
Method names in an Android context	10,693,844 60.8%	5,477 4.5%
Method names in an app context	2,712,754 15.4%	99,240 82.3%
Method names in an API context	4,192,055 23.8%	19,849 16.5%
API method names	637,246 3.6%	6,344 5.3%
Method names total	17,598,653	120,621
Exceptions in Android context methods	355,925 26.8%	100 21.6%
Exceptions in app context methods	281,684 21.2%	230 49.8%
Exceptions in API context methods	692,490 52.1%	314 68.0%
Root cause exceptions	901,274 67.8%	406 87.9%
Chained exceptions	726,655 54.6%	316 68.4%
Exceptions total	1,330,099	462

4 Results and Analysis

Our findings can be analyzed at two levels. At a primary level we can observe common problematic API methods, reported exceptions, and exception chaining practices. Analyzing further our data we want to see *why* these crashes occur: what particular API design problems caused the software crashes we observed?

4.1 Primary Observations

Key metrics regarding the stack traces we analyzed appear in Table 4. The left-column metrics measure each value's occurrence within the stack traces (**total**), while the right-column ones measure the **unique** occurrences of a value—a class, method, or exception name. As a particular value can occur in multiple layers this column's percentages can add to more than 100%.

In total we analyzed almost 900 thousand stack traces containing 16.7 million method calls from 120 thousand methods and more than one million exceptions. An

[7] http://stackoverflow.com/
[8] http://code.google.com/p/android/issues/list

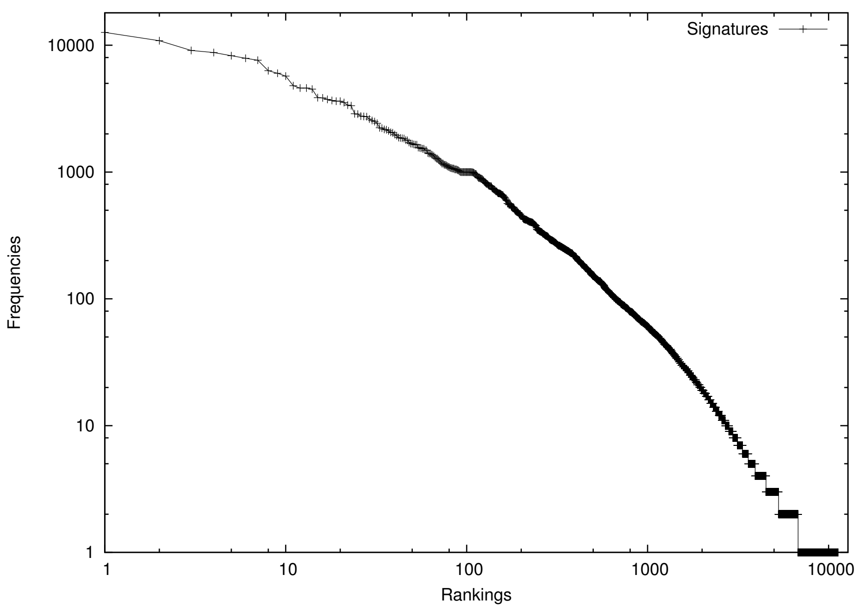

Fig. 3: Distribution (log scale) of the ranked signatures (total frequencies).

exception's stack trace contains on average 19 method calls, a sign of a complex layered architecture. On average each stack trace appears in our data set six times, but behind this average lies a very uneven distribution. From the analyzed stack traces (**total**) we found a half million (531,432) with API methods and exceptions associated with the Android framework. The first four most common entries appear 17,074, 12,637, 10,864, and 9,100 times, whereas the entry at the 75% quartile appears three times, and the one at the 50% quartile just one time. Figure 3 shows the distribution of the signature frequencies. After the top 600 signatures the frequencies decrease significantly and the corresponding crashes are associated either with less commonly used methods or with programming errors.

Observation 1: Distribution of method and class names

By applying the techniques we described in Section 3.3 we were able to separate classes, methods, calls, and exceptions among the Android framework, the applications, and diverse APIs. As expected, most of the unique method names (82.3%) appear in the application context, mirroring the large relative size of the application code. The next most common set of unique method names are those corresponding to API implementation (16.5%), again illustrating that the API implementation is richer than the part of the framework that launches and calls the applications, whose unique methods correspond to 4.5% of the total. (We regard a call from the application to a framework function as an API call.) We located 6,344 unique API method names, indicating an average API call depth of three in our data set. The

Table 5: Total number and % of method calls among layers.

From / To	Android	app	API
Android	9,829,209	755,465	10,314
	58.9%	4.5%	0.1%
app	0	1,812,243	637,246
	0.0%	10.9%	3.8%
API	0	108,474	3,544,428
	0.0%	0.6%	21.2%

distribution of method names among the total (non de-duplicated) stack traces is completely different from the unique ones having 60.8% of the names appearing in the Android framework, 15.4% in the application, and 23.8% in the API layer. This illustrates the common and deep Android origin of all stack traces and the relatively shallow call paths along the application and API levels. Class names are distributed similarly to method names, showing that the correspondence of methods to classes is comparable among the three layers.

Observation 2: Distribution of exceptions

The most interesting part of our key metrics concerns the exceptions. Most exceptions (52.1%) appear at the context of API methods indicating that there is a lot of value associated with improving the stability of API calls. Furthermore, the very small number of API method names associated with these exceptions (3.6%) further increases the value that can be gained through a better design of these interfaces. The second most common site of exceptions is Android's framework, showing that the framework's developers can do a lot to improve the stability of the applications running under it. Finally, exceptions directly within an application account for 21.2% of the total number of exceptions.

Observation 3: Categorization of calls among layers

The categorization of calls appearing within the analyzed stack traces across the three layers appears in Table 5. These represent a summarized run-time version of the call graph, according to the observed calls. It is apparent that most of the activity happens within the layers (along the Table's diagonal), with Android doing most of the heavy lifting (58.9% of the calls). The application to API calls represent 4.5% of the observed calls.

The **unique** number of method calls in Table 6, represent a static version of the call graph, which corresponds to the implementation structure of the code running on Android phones. Here most interactions are within applications (37.5%), with calls from application to API methods representing the second highest percentage (22.3%) indicating the importance of API methods in an application's operation.

Observation 4: Top API method exceptions

The exceptions that the API methods report to applications (Table 7) tell a sad story. One third of the exceptions are of the most generic kind (`RuntimeException`),

Table 6: Unique number and % of method calls among layers.

From / To	Android	app	API
Android	9,059	38,986	105
	4.6%	19.7%	0.1%
app	0	74,160	44,127
	0.0%	37.5%	22.3%
API	0	5,585	25,955
	0.0%	2.8%	13.1%

Table 7: Top ten API-reported exceptions.

Exception	%
java.lang.RuntimeException	31.6
java.lang.OutOfMemoryError	12.2
java.lang.NullPointerException	7.5
java.lang.IllegalArgumentException	6.9
java.lang.IllegalStateException	3.9
android.view.WindowManager$BadTokenException	3.4
android.view.InflateException	2.9
java.io.FileNotFoundException	2.2
android.database.sqlite.SQLiteException	1.9
android.content.ActivityNotFoundException	1.8
Total	**74.4**

Table 8: Top ten chained exception causes.

Reported exception	Root exception	%
java.lang.RuntimeException	java.lang.OutOfMemoryError	20.2
java.lang.RuntimeException	java.lang.NullPointerException	16.7
java.lang.RuntimeException	java.lang.IllegalArgumentException	7.1
android.view.InflateException	java.lang.OutOfMemoryError	6.9
java.lang.RuntimeException	android.database.sqlite.SQLiteException	5.9
java.lang.RuntimeException	java.lang.IllegalStateException	3.6
java.lang.RuntimeException	android.os.DeadObjectException	2.5
java.lang.RuntimeException	android.database.sqlite.SQLiteDiskIOException	2.2
java.lang.RuntimeException	android.content.res.Resources$NotFoundException	1.7
java.lang.RuntimeException	java.lang.SecurityException	1.6
Total		**68.3**

which surely doesn't help a developer get to grips with the exception's cause. This is further illustrated in Table 8, where in most cases diverse API failures appear to be propagated up as a `RuntimeException`.

Observation 5: Top root exceptions

Looking behind the scenes at the root exceptions behind the application crashes (Table 9), we see that 29.2% of the crashes occur due to erroneous object reference handling, throwing `NullPointerException`. Also, memory handling in An-

Table 9: Top ten root cause exceptions.

Exception	%
java.lang.NullPointerException	29.2
java.lang.OutOfMemoryError	14.2
java.lang.IllegalArgumentException	6.6
java.lang.RuntimeException	4.5
java.lang.IllegalStateException	4.1
android.view.WindowManager$BadTokenException	2.6
android.database.sqlite.SQLiteException	2.6
java.lang.IndexOutOfBoundsException	2.1
java.lang.ArrayIndexOutOfBoundsException	1.9
java.io.FileNotFoundException	1.7
Total	**69.5**

droid's applications is causing a great percentage of problems (14.2%), throwing `OutOfMemoryError`.

Observation 6: Top API methods and crashes

In Table 10 we can see that just 10 API calls result in 16.7% of the crashes. Finally, Table 11 shows the most popular API methods. We see that there are methods with thousands of calling methods, which indicates that improvements to the design of these API methods will benefit many applications.

4.2 Crash Cause Categories

In this section, we present a categorization for the most popular causes of Android application crashes. For this, we analyzed signatures from stack traces and tried to identify major groups of crash causes according to the following process.

1. Using the method we described in Section 3.3, we located 11,277 signatures with API methods. From these signatures we found 328 with invalid exceptions (`UnknownException`). The located signatures came from 531,432 total and 65,425 unique stack traces; the remaining stack traces were associated with application-specific methods and exceptions.
2. We sorted the 11,277 signatures based on their total number of occurrences and we divided the sample into three subgroups depending on the packages of the included API methods, namely: `(com.)android.*`, `java.*`, and `com.*`. We created these subgroups in order to compare APIs from different sources—but associated with the same framework—and generalize our results.
3. For each subgroup, we examined the participated signatures, trying to understand the crash causes of the associated stack traces. According to Sullivan and Chillarege (1991); Ganapathi et al (2006) though, the identification of root causes in software crashes is not as straightforward as in bugs. This occurs because the source code of the crashed software is not always available and we cannot guess all the environment parameters that probably contributed to a crash. In our case, for the majority of the stack traces we were able to find the crash causes only

Table 10: Top ten API method crashes.

API method	Reported exception	%
android.app.Activity.-setContentView	java.lang.RuntimeException	4.3
android.app.Dialog.dismiss	java.lang.IllegalArgumentException	2.2
android.view.LayoutInflater.-inflate	android.view.InflateException	1.6
android.app.Activity.-startActivity	android.content.ActivityNotFoundException	1.5
android.graphics.-BitmapFactory.-decodeResource	java.lang.OutOfMemoryError	1.4
android.app.Dialog.show	android.view.WindowManager$BadTokenException	1.4
com.android.internal.view.-BaseSurfaceHolder.-unlockCanvasAndPost	java.lang.IllegalArgumentException	1.1
android.graphics.Bitmap.-createBitmap	java.lang.OutOfMemoryError	1.1
java.util.ArrayList.get	java.lang.IndexOutOfBoundsException	1.1
android.view.LayoutInflater.-inflate	java.lang.RuntimeException	1.0
Total		**16.7**

Table 11: Top ten API methods with the most callers.

Method	# callers
android.app.Activity.setContentView	2112
android.app.Activity.startActivity	1450
android.view.LayoutInflater.inflate	1205
android.app.Dialog.show	1187
java.util.ArrayList.get	1101
android.app.Dialog.dismiss	1061
android.os.AsyncTask.execute	552
android.graphics.Bitmap.createBitmap	525
android.app.Activity.showDialog	483
android.content.res.Resources.getDrawable	457

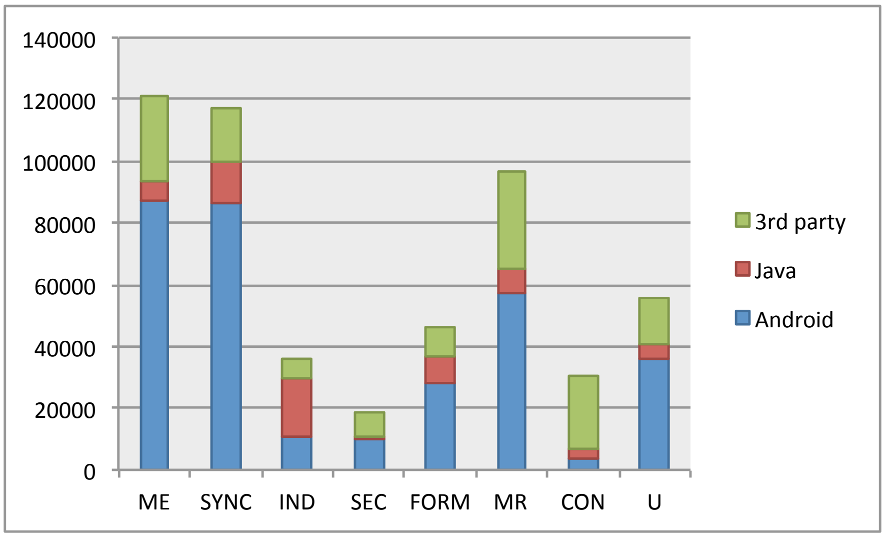

Fig. 4: Causes of API-related crashes (**total** occurrences).

from the names of the exceptions (e.g. `OutOfMemoryError` refers to memory exhaustion). However, a significant number of the signatures had generic unchecked exceptions (see Table 1 and Section 4.2.8), which made the crash causes unclear. For these signatures, we studied the Android API reference,[9] and consulted Q&A sites, such as *stackoverflow.com*.

4. Knowing the crash causes for all signatures, we grouped them in representative crash cause classes. As we had no previous knowledge of these classes, we concluded on them, gradually, as we were studying the signatures. Figure 4 shows the categories for the total instances and Figure 5 for the unique instances of the stack traces. As we can see, the distribution of the signatures among the subgroups and categories are similar for both diagrams.

Thus, we mapped the space of the main reasons that application failures occur on mobile devices that run on the Android platform. In Section 6, we present threats to validity regarding our sample. In the following, we discuss each crash category giving examples from representative signatures allocated to them (see Table 12). In particular, Table 12 shows examples from signatures and their frequencies in the total and unique samples. Finally, we discuss our findings for the three subgroups of the APIs: Android, Java, and third-party libraries. In the Appendix we illustrate the distributions of the signatures for each of the examined APIs.

4.2.1 Memory Exhaustion (ME)

We found that the most common application crash cause is related to memory leaks. This is complementary to the findings of Tan et al (2013). This was a result we ex-

[9] `http://developer.android.com/guide/components/index.html`

Table 12: Signatures

#	Method Application Exception Root Exception	Total	Unique
1	android.view.LayoutInflater.inflate android.view.InflateException java.lang.OutOfMemoryError	10864	1635
2	android.view.View.setBackgroundResource java.lang.OutOfMemoryError java.lang.OutOfMemoryError	1400	370
3	android.database.sqlite.SQLiteOpenHelper.getReadableDatabase android.database.sqlite.SQLiteDatabaseLockedException android.database.sqlite.SQLiteDatabaseLockedException	1640	188
4	android.os.AsyncTask.execute java.util.concurrent.RejectedExecutionException java.util.concurrent.RejectedExecutionException	1561	236
5	android.app.Dialog.dismiss java.lang.IllegalArgumentException java.lang.IllegalArgumentException	12637	1280
6	java.util.ArrayList$ArrayListIterator.next java.util.ConcurrentModificationException java.util.ConcurrentModificationException	2218	196
7	org.jaudiotagger.audio.AudioFileIO.read org.jaudiotagger.audio.exceptions.InvalidAudioFrameException org.jaudiotagger.audio.exceptions.InvalidAudioFrameException	4799	70
8	android.app.Activity.setContentView android.view.InflateException java.lang.NullPointerException	2760	390
9	android.app.Activity.startActivity android.content.ActivityNotFoundException android.content.ActivityNotFoundException	9100	1453
10	java.util.ArrayList.get java.lang.IndexOutOfBoundsException java.lang.IndexOutOfBoundsException	7626	1276
11	android.app.Activity.startActivity java.lang.SecurityException java.lang.SecurityException	1872	46
12	com.google.api.services.tasks.Tasks$Tasklists$Get.execute javax.net.ssl.SSLException javax.net.ssl.SSLException	115	11
13	java.text.DateFormat.parse java.text.ParseException java.text.ParseException	149	12
14	android.database.sqlite.SQLiteDatabase.execSQL android.database.sqlite.SQLiteException android.database.sqlite.SQLiteException	1642	219
15	org.apache.http.impl.client.AbstractHttpClient.execute org.apache.http.conn.ConnectTimeoutException org.apache.http.conn.ConnectTimeoutException	659	32
16	android.hardware.Camera.open java.lang.RuntimeException java.lang.RuntimeException	3862	144
17	android.content.ContentResolver.insert android.database.sqlite.SQLiteException android.database.sqlite.SQLiteException	314	18
18	android.graphics.Canvas.drawBitmap java.lang.NullPointerException java.lang.NullPointerException	628	79

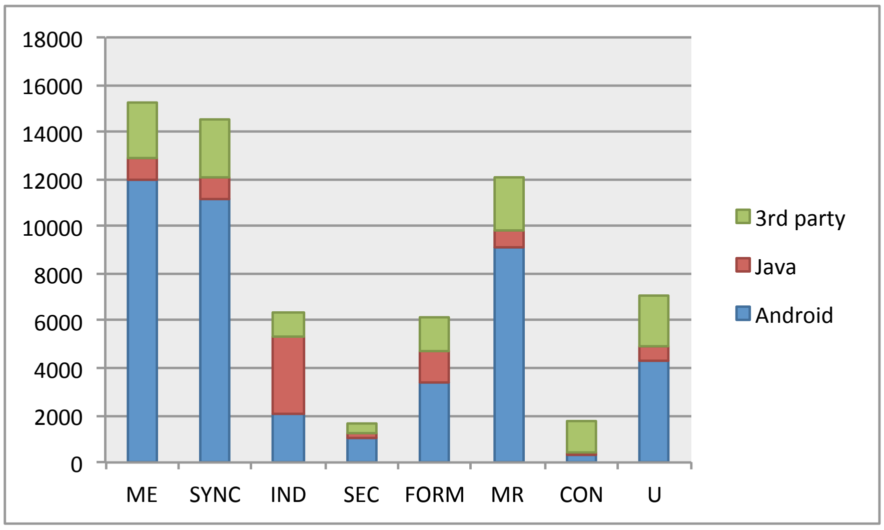

Fig. 5: Causes of API-related crashes (**unique** occurrences).

pected, as most mobile devices have constrained memory and developers are seldom aware of the amount of the available memory. Both diagrams above (Figure 4 and Figure 5) show that memory exhaustion is the dominant crash cause (almost 30%) in Android APIs. This percentage is similar to the findings of Sullivan and Chillarege (1991) and close to the results of Tan et al (2013). The latter imply that memory bugs is a leading root cause in crashes (greater than 38%). The allocation of signatures to this category was straightforward, as the presence of the `OutOfMemoryError` helps one to distinguish those related to memory exhaustion. Examples 1 and 2, in Table 12, are characteristic of this category. The first example is related to a failed import operation for a bitmap object and the second to a failed attempt to load many bitmap objects at once.

4.2.2 Race Condition or Deadlock (SYNC)

The second most common cause of application crashes is associated with race conditions and deadlocks. Again, we found that this cause of crash is the top (27%) in Android APIs. In addition, we found that Java APIs count a significant number (20%) of such crashes, possibly because many developers use Java threads wrongly. As Android is not an application, but a device that hosts applications, our findings agree with the results of Tan et al (2013) for the Linux kernel and the significance of concurrency bugs in operating systems. We identified the signatures for this category based on the types of their exceptions and the associated API documentation. There are many types of synchronization problems; this category contains signatures from Table 12 related to: a. database deadlocks (example 3), b. race conditions in asynchronous tasks (example 4), c. abnormal execution of the lifecycle of an activity (example 5), and d. synchronization issues with iterators (example 6).

4.2.3 Missing or Corrupt Resource (MR)

A great number of crashes occur because of missing or corrupt resources. In this category, we have added signatures that imply the absence of a resource or the inability of the system to decode a resource. In particular, we refer to external resources, such as an image or an audio file. We found many of such problems mainly in Android and third-party APIs (20% for each). In Table 12, examples 7 and 8 are representative of this category. The names of the exceptions, however, do not indicate, clearly, resource problems. We concluded that such signatures belong in this category, by examining their stack traces, the documentation of the exceptions, and posts in *stackoverflow.com*. In this category, we have also allocated signatures that indicate crashes due to either undeclared components or the system's failure to locate a suitable component for a specific task. It was easy for us to identify such signatures, as the Android's API provides characteristic exceptions for components, such as the `ActivityNotFoundException`. For instance, consider example 9, in Table 12. There are two possible reasons behind this crash: either the developer has forgotten to declare the activity in the `AndroidManifest.xml` file or the developer does not call the component with the right identifier.

4.2.4 Indexing Problem (IND)

Crashes due to indexing problems can be caused by invalid loop conditions and inappropriate structures. As it was expected this crash cause is dominant (almost 40%) in Java APIs, because of the common use of Java packages (`java.lang.*`, `java.util.*`) for the manipulation of data structures. Example 10 in Table 12 shows a signature from such a crash.

4.2.5 Insufficient Permission (SEC)

Here, we have mainly allocated signatures with security exceptions. We found security issues mainly in Android and third-party APIs. For instance, consider the representative example 11 in Table 12. This example reflects that there is an activity permission problem in the `AndroidManifest.xml` file. Because of this problem, the activity cannot start, as the `Intent` object, which should be passed to the system, has not got the right permissions (e.g. for another device to be eligible to receive a message.) Also, consider example 12 in Table 12 from a third-party API and a failed attempt of a service to open due to a security reason. We assume that insufficient documentation could be a main problem for the above examples (see also example 16 in Table 12 and 4.2.8).

4.2.6 Invalid Format or Syntax (FORM)

This category refers to crashes due to erroneous method inputs and their number is significant (17%) in Java APIs. Considering that many of such crashes were caused by typos and careless programming, the above result matches a similar finding (15%) of Tan et al (2013). Specifically, the signatures that belong here imply format problems and invalid syntax of SQL queries. For instance, the exceptions of example 13 in Table 12 indicate that the crash caused either because the input value was `null` or the format of the input was invalid. In addition, the exceptions of example 14 in Table 12 shows that the signature is related to wrong SQL query syntax.

4.2.7 Connectivity Problems (CON)

In this category, we have allocated signatures associated with networking exceptions, such as the `ConnectTimeoutException`. These problems appear especially in third-party libraries (10-20%) concerning the connectivity with external directories and services. In example 15 (Table 12), this exception reflects a timeout, while connecting to an HTTP server, which sends a `null` message.

4.2.8 Unclassified (U)

Signatures that belong to this category do not give clear information about the real causes of their crashes. Such cases were especially common in Android and third-party APIs (10-15%), meaning that they have a lot of undocumented exceptions. In the following, we present some examples to make our argument evident. In Table 12, example 16 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 unchecked `RuntimeException` is generic for one to understand whether the crash cause is a race condition or an insufficient permission. Likewise, in example 17 (Table 12), which is related to a database issue, the exceptions (`SQLiteException`) do not reveal the real crash cause. There may be several reasons behind this crash. There would be an invalid insert statement, a synchronization, or a connectivity problem. Finally, in example 18 (Table 12), it is unclear whether the exception (`NullPointerException`) was thrown because of a race condition (device orientation change) or a missing resource (careless declaration).

5 Crash Mitigating Recommendations for APIs, Tools, and Frameworks

In this section, we present an overview of API recommendations, development tools, and frameworks that can improve APIs' design and implementation and help developers to avoid application crashes. The mentioned crashes are associated with the categories in Section 4.2. API recommendations refer to API design and implementation choices. The suggested tools refer to ad-hoc software that developers can use, during implementation, to test their applications. Finally, we present some execution platform and framework designs that can be used to protect the running applications from unexpected crashes.

5.1 API Recommendations

Here, we discuss indicative API recommendations that could solve problems associated with the crash categories listed in Section 4.2. These decisions refer to the design of the interfaces and their implementation. We base these API recommendations on a qualitative study. Specifically, we examined the signatures of our sample, we looked at related parts in the Android API reference, and made suggestions based on our experience after iterative discussions. For the presentation of our suggestions, we use as reference the representative signatures of Table 12 and the observations of Section 4.1. Finally, we provide the frequencies of the representative signatures to show how many crashes could be avoided based on the following solutions.

5.1.1 Memory Exhaustion

As we mentioned in Section 4.2, common crash causes in mobile software are related to memory exhaustion (also consider the `OutOfMemoryError` in the tables of Observation 4, Observation 5, and Observation 6.) A representative example of such crashes involves the case when a resource cannot fit in the available memory (see the example 1, in Table 12). In particular, consider the process of loading a new file into memory. Often developers have to load a resource before decoding it. Therefore, they are responsible to decide, in advance, for the size of the resource (e.g. an image's resolution). Most of the times though, developers have scant information regarding the available memory (Yang et al 2004). Thus, it is possible for the memory to be exhausted. To avoid these problems, the system itself could check and tailor the size of the resources to be loaded. Therefore, the API could provide a **resource auto-resize interface**—consider image compressive sampling (Candes and Wakin 2008).

Things become complicated when developers have to load more than one memory consuming object at once (example 2, in Table 12); this is often the case when they build the GUI, which consists of many associated bitmaps, and it needs recycling and reloading quite often. In such cases, developers can use structures, which operate as caches (e.g. the Android's `LruCache`). As we argue, however, below, this choice has some drawbacks.

Cache structures may accelerate the access to recently used objects, but increase consumption memory. Consequently, the API should not support cache structures of an arbitrary size but **restricted cache structures**. In addition, this problem can be solved through the use of appropriate algorithms by the garbage collector (Bond and McKinley 2008).

Furthermore, some file formats, especially for images, may require a lot of memory, when they are uncompressed, being susceptible to memory leaks. There are, however, less memory consuming formats one can use instead. For instance, vector graphics (`.svg`) can be lighter than bitmap image files (`.png`, `.jpg`) (Vaughan-Nichols 2001). Hence, it would be beneficial for a system if its API includes **memory efficient file formats**. Ultimately, the system itself could automatically convert expensive formats into memory efficient ones—when it is necessary. Nevertheless, some devices cannot support particular formats yet; see Table 2 in Gavalas and Economou (2011).

Finally, the over allocation of heap memory, for a particular process, can result in out of memory errors. When the system allows developers to increase the heap memory—as Android does with android:largeHeap="true"—they will burden the memory quite often. Consequently, the system itself—not the developer—should decide when there is a need for extra heap memory (**fixed heap memory**).

5.1.2 Race Condition or Deadlock

According to our categorization in Section 4.2, synchronization problems are common causes of application crashes in mobile software. This is also obvious from the rankings of relevant exceptions (`IllegalStateException`, `BadTokenException`) in the tables of Observation 4, Observation 5, and Observation 6. These results are reasonable as developers can easily misstep when building multithreading applications. Therefore, systems themselves should provide functional APIs that prevent developers from writing error-prone programs.

In order to avoid deadlocks, the APIs should give developers the capability to write **non-blocking algorithms** (Michael and Scott 1996); as Intel does with the thread building blocks (Pheatt 2008). These algorithms are not based on locks (`synchronized` methods, such as in example 6, in Table 12), but on atomic hardware primitives and generic algorithms; consider the *compare-and-swap* by Valois (1995). This is particularly important in mobile applications where the main thread (GUI) should response instantaneously to events triggered by the user and device orientation changes (see example 5, in Table 12.)

Keeping the main thread lock-free is useful for two reasons: to isolate long-lasting operations (see example 3 in Table 12), and to shield the main thread against an abnormal sequence of events. Android, for instance, promotes an API design—`AsyncTask` class—that keeps the main thread lock-free ("Single thread model").[10]

Finally, in order to keep the interactions with the main thread under control (see example 4 in Table 12) and the execution of the events in a sequence, the API can provide a non-blocking **buffer mechanism** (Kim 2006).

5.1.3 Missing or Corrupt Resource

Application crashes can occur when the system is unable to process a specified resource (image, audio file, even a class), because it is missing or corrupt resource. A practical way to avoid such crashes is the adoption of **default resources** by the API. For instance, when a screen background layout is missing or corrupt (see example 8, in Table 12 and `InflateException` in Observation 6), the system can open the default one. In addition, **specific exceptions** associated with specific missing resources can help developers to locate, directly, the problematic resources. More importantly, the API design can force the statically checked binding of resources with an application. Then, the application will refuse to crash when the resources are missing or corrupt.

Also, to limit the number of crashes because of flawed declaration of application components (see example 9, in Table 12 and `ActivityNotFoundException` in Observation 6) the API can provide appropriate **type checking** of resources and **meaningful component codes**. For instance, in Android, to start a new activity, the programmer has to use an `Intent` object, by providing either a known component identifier or a description of a task to be accomplished. When the description is provided, the system is responsible to choose by itself the most suitable component for that task. If the system fails to find it, it crashes. However, type checking can ensure the existence of an activity intent before execution. The association with type component identifiers can, also, prevent developers from the careless use of component identifiers.

[10] `http://developer.android.com/guide/components/processes-and-threads.html`

5.1.4 Indexing Problem

To decrease the number of indexing problems (consider the percentages of the root exceptions in Observation 5), the API can provide structures with error-free arguments and appropriate error handling.

Error-free arguments imply iterators instead of integer indices and implicit loops.

Error ignorance, also, is an API design that can allow the system to ignore specific errors. For instance, when the threshold of a condition, in a loop, is greater than the number of the elements in an array, the system does not need to throw an out of bounds exception.

Finally, in order to avoid indexing problems with database cursors, the API can establish a **bound** for the number of the rows that will return from the database (see example 10, in Table 12).

5.1.5 Invalid Format or Syntax

Many crash causes occur because of trivial errors in the format or syntax of method arguments (see example 13 and 14, in Table 12).

To decrease the number of crashes due to an invalid query syntax, the API can include an **interface for queries on collections**. An example of such an interface is LINQ (Meijer et al 2006). Also, the API should use **domain specific data types** (Mernik et al 2005) rather than primitive ones or string, as the former offer more possibilities for static checking.

5.1.6 Insufficient Permission

The system itself should check (through an `Intent` object) if a given permission is valid or not, and throw **specific exceptions**. For instance, example 16, in Table 12 can refer to a crash because of a missing permission regarding the opening of the camera by an application. However, the thrown exception (`RuntimeException`) is generic for one to understand the cause of the crash and fix the problem. Also, in Android, the system could inform the programmer about missing permissions, on the fly, when he or she declares new components into the `AndroidManifest.xml` file (see example 11, in Table 12).

5.1.7 Connectivity Problem

To eliminate crashes related to connectivity problems (see example 15, in Table 12) the system apart from throwing exceptions can, also, 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. Another possible solution for avoiding problems due to connectivity errors includes postings from the system regarding the network's connectivity.

5.1.8 Unclassified

As we mentioned in Section 4.2.8, generic unchecked exceptions do not help developers to understand directly what the cause behind a crash is. According with Henning (2009), a well-designed API should provide clear information about each method and relevant explanations for the exceptions that can occur. However as Observation 4 shows, one third of the API exceptions in our signatures had generic exceptions (`RuntimeException`). Consequently, the API should use **specific exceptions** depending on the problem and provide documentation with related examples.

5.2 Development Tools

In this section, we present an overview of tools that developers can use during the implementation of an application to avoid API-related crashes. Specifically, such profiling, testing, and static checking tools can locate programming errors (**fault localization**) and avert common application crashes (**fault prevention**). Searching in the bibliography, we focused on tools dedicated to Android software. We consider the use of such tools significant, before the release of an application, as in many cases performance issues related to resource management (e.g. memory and concurrency issues) are not documented in the API reference to inform developers in advance. In the following sections, we present a short survey of tools that can prevent the crashes in the corresponding classes of Section 4.2.

5.2.1 Profiling Tools

Profiling tools can support memory, process, and network traffic analysis. In the following, we have categorized such tools based on the analysis they conduct.

Memory analyzers give a picture of the allocated objects into the heap, over a period of time, and find memory exhaustion problems. For instance, consider: the Tracker tab (provided by the Dalvik Debug Monitor Server—DDMS),[11] the AMOS (Seo et al 2011) runtime memory fault detection tool, the Eclipse Memory Analyzer (MAT),[12] and the JHat.[13] Also, there are tools that visualize the memory consumption of Java applications, such as the Heapviz that helps developers to navigate large, pointer-based data structures (Aftandilian et al 2010) and the Eclipse plug-in by Alsallakh et al (2012) that supports the visualization of arrays and collections during debugging (for the identification of indexing problems).

Thread debuggers are execution log viewers that identify race conditions and deadlocks. For example, consider: Traceview[14] (comes with the DDMS) and Systrace,[15] as well as JProfiler,[16] Optimizeit[17] and Jinsight.[18] The latter tools pro-

[11] `http://developer.android.com/tools/debugging/ddms.html`
[12] `http://www.eclipse.org/mat/`
[13] `http://docs.oracle.com/javase/6/docs/technotes/tools/share/jhat.html`
[14] `http://developer.android.com/tools/debugging/debugging-tracing.html`
[15] `http://developer.android.com/tools/help/systrace.html`
[16] `http://www.ej-technologies.com/products/jprofiler/overview.html`
[17] `http://docs.oracle.com/cd/E19830-01/819-4721/beafp/index.html`
[18] `http://www-03.ibm.com/systems/z/os/zos/features/unix/tools/jinsightlive.html`

vide a visual representation of the virtual machine and show the garbage collector's activity. They can locate both memory exhaustion and synchronization problems.

Network analyzers identify and analyze connectivity problems. For instance, Android provides a network traffic tool and a tool for tracking the network state changes. Both tools are included into the DDMS.[19] Also, Netflix's Chaos Monkey,[20] is a cloud testing tool, for implementing resilient services on the cloud, as well as the FSaaS method (Faghri et al 2012).

5.2.2 Testing Tools

Apart from the JUnit framework,[21] developers can use tools for testing and verification to identify race conditions and deadlocks.

GUI testing is associated with several types of tools for the identification of synchronization problems. There are tools based on regression testing (production of streams of pseudo-random user events) such as "Monkey"[22] and A^2T^2 (Amalfitano et al 2011, 2012). There are also other tools based on computer vision, such as the Sikuli testing tool (Chang et al 2010).

Runtime verification is useful for locating deadlocks. A well-known model checking tool is the Java PathFinder (Havelund and Pressburger 2000). For Android applications, there is the JPF-Android tool, which is based on the Java PathFinder (van der Merwe et al 2012). Another tool for monitoring the execution of Java programs, is the Java PathExplorer (Havelund and Roşu 2004). Finally, there are tools that can test particular inputs (concolic testing and constraint satisfaction), such as: ConAn (Long et al 2003), jCUTE (Sen and Agha 2006), and LCT (Kähkönen et al 2011).

5.2.3 Static Checking Tools

Static checking examines the source code without program execution. Thus, developers can use automated tools to check the quality of their applications before execution. In the following, we present static checking tools that can detect problems related to several API crash categories (see Section 4.2).

Static deadlock detectors can identify race conditions and deadlocks. Dimmunix (Jula et al 2008, 2011) is such a tool for Android applications. Also, RacerX (Engler and Ashcraft 2003), Jlint (Artho and Biere 2001), and EPAJ (Agarwal et al 2006) find multithreading problems in Java programs.

Static code analyzers test applications from many different angles. FindBugs can detect more than 300 coding errors, such as null pointer problems, array overflows, and bad name conventions (Hovemeyer and Pugh 2004; Ayewah et al 2008). Moreover, Julia, which is a static analyzer for Android applications, checks equality, class casts, bad programming styles, dead code, method redefinitions, nullness, and code termination (Payet and Spoto 2012). Especially through nullness

[19] http://tools.android.com/recent/detailednetworkusageinddms
[20] http://www.linuxinsider.com/story/75780.html
[21] http://junit.org/
[22] http://developer.android.com/tools/help/monkeyrunner_concepts.html

analysis, developers can avoid inflate exceptions (thrown due to missing or corrupt resources) related to the declared resources (e.g. layouts) in the XML file, namely. The Android lint tool[23] also lets developers analyze XML files, and find inefficiencies in the hierarchy of the GUI `Views`.

Code snippets are blocks of reusable source code. As APIs tend to be complex, sometimes it is hard for the developers to understand them and find quickly appropriate code snippets. Prospector is a plug-in for Eclipse that assists developers to find the features they are looking for (Mandelin et al 2005). Such tools can help developers to choose the right interfaces for implementation and, consequently, prevent applications from crashes. In addition, MAPO is a searching tool for code snippets that uses API methods of interest, which can help programmers to reduce coding bugs (Xie and Pei 2006).

5.3 Execution Platforms and Frameworks

Given the categories in Section 4.2, we searched for execution platform and framework designs that can make a system resistant to crashes. The crashes we mention, here, are related to our two top categories: memory exhaustion and race conditions and deadlocks. In the following, we present frameworks associated with the Android platform. However, the highlighted techniques can be adopted by similar systems.

5.3.1 Process Management

Process management depends on techniques and algorithms that an operating system uses. Some platforms and frameworks, however, can get a system's capabilities beyond the conventional and shield applications from crashes. The Aciom framework, for instance, identifies application I/O requests according with their criticalness and applies **request priorities** (Kim et al 2011b). Thus, it can prevent time-sensitive applications from delays, averting the system from race condition or deadlock crashes. In addition, the platform of Chen et al (2011), which is based on **remote resource management**, can make applications require less memory and resources. Hence, it can eliminate the well-known "non-responsive" exceptions in Android. MetaService also is an interface that can enable the **object transferring** between Android applications and abolish the need for object processing (marshaling/unmarshaling) and replication (Choe et al 2011). Thus, it can help avoid crashes due to race condition or deadlock and memory exhaustion. Finally, the capabilities of hosted operating systems can be improved by tuning. See common techniques in the work of Bovet and Cesati (2005), regarding: **prioritization**, **scheduling**, **freezing**, and **swapping**. Process virtual machines, also, may need tuning (Schoeberl 2004; Maia et al 2010) to satisfy the requirements of the running applications (real-time responsiveness).

5.3.2 Memory Management

To avoid memory exhaustion, constrained systems should use the heap space efficiently (Panda et al 2001). This is feasible with the use of appropriate memory

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

management techniques and garbage collection algorithms. To keep small heaps under control and avoid fragmentation (Bacon et al 2004), a garbage collector should perform **compaction** (Maia et al 2010). For instance, consider the "mark-compact" garbage collector of Chen et al (2003), which is based on object compression and lazy allocation management. An **automatic heap-sizing** algorithm can be also used by garbage collectors to determine *a priori* the appropriate heap size, and minimize paging problems (Yang et al 2004). Memory capacity can be also increased through efficient software-based RAM **compression** (Yang et al 2010). In addition, as multimedia applications make significant use of arrays, it is necessary for systems to manipulate them efficiently. For this, Fraboulet et al (2001) have developed an optimal algorithm to reduce the use of temporary arrays (`LruCache` collection in Android). They used a technique called **"loop fusion"** (McKinley et al 1996). Finally, according to Schoeberl (2004), designers of constrained embedded systems should pay attention on the memory that the libraries consume (e.g. Android's `Zygote`). This is important in order to leave more memory to the applications. Thus, the use of **lightweight and compact hosted systems** can help avoid out of memory crashes.

6 Threats to validity

The possible limitations of our study's empirical results (the crash causes we identified and the recommendations we proposed) concern: a) internal validity i.e. whether the results are valid *per se*, and b) external validity i.e. whether the results apply to other platforms and frameworks.

6.1 Internal validity

As we had no *a priori* knowledge of the methods that belong to the Android framework, we used heuristics to determine them (see Section 3.3). Briefly, we isolated n-tuples (that consist of method names) from stack traces and determined their frequency. Then, by ordering the n-tuples based on their frequency, we found the most common ones and established the name space of the Android framework's methods. In addition, we examined manually other common n-tuples to find application-specific methods, and this possibly implies a selection bias. Thus, 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 libraries.

6.2 External validity

External validity aims to ensure that the findings of our empirical study can be generalized for other samples too. This, however, can be achieved only through the replication of our study on different samples from Android, as well as other similar platforms. We believe, though, that our findings are representative for a large population for a number of reasons.

A first threat to external validity might be that for another platform (e.g. iOS or Windows Mobile) our categories would be different. Nevertheless, there are three reasons for which our results can be generalized: 1) the large amount of the crashes and applications we examined, 2) the significant diversity of the devices that the Android platform runs on, and 3) the extent of the Android's API. Finally, our categories refer to quite common crash causes that occur in constrained devices and applications written in object oriented programming languages (Java, Objective-C, and C#).

A second threat to external validity might be related to the generalization of our API design and implementation recommendations for mobile device embedded systems, based on a sample coming from a specific platform (i.e. Android). We believe, however, that the fact that Android is currently the leading platform for mobile applications and devices, makes the design and implementation decisions for its API to have an impact on its competitors. Given that an application can be written for Android, iOS, and probably Windows Mobile, the basic design concepts of these platforms' APIs should be similar. Thus, even though we need to empirically validate our assumption, we argue that these indicative recommendations can be considered universally applicable.

Finally, we acknowledge that the data set we used came from specific applications chosen by our provider, BugSense, and applications that had agreed to send crash reports. This could make our sample biased. However, the amount of the different crashes even for a small sample of applications can give a real picture of common crash causes. As future work, we would like to analyze data from more applications and sources. In addition, we recognize that our sample contains three months of data, which is a short period of time. Even though our data set comprised a large number of crashes from different API versions, we are committed to examine more data from different time windows.

7 Conclusions and Future Work

In this paper we used software telemetry data from Android application crashes to see how crashes are associated with API deficiencies. We processed approximately a million stack traces to pinpoint critical API calls by extracting representative signatures, ordering the signatures according to their frequency, investigating the crash causes associated with our signatures, and categorizing the causes into eight classes.

Our findings show that the top crash causes can be attributed to memory exhaustion, a race condition or a deadlock, or a missing resource. However, we were unable to classify the crash causes for a significant number (more than 10%) of signatures due to the generic exceptions associated with them (`RuntimeException`, `NullPointerException`) and inadequate API documentation. These results can help in the design of more resilient APIs and (for particular crash causes) drive improved implementation.

In addition, we suggested ways to reduce the number of application crashes associated with API deficiencies by making API design and implementation recommendations for each crash cause category. We argued that more specific exceptions, non-blocking algorithms, and default resources can eliminate the most frequent crashes. We also suggested that development tools like memory analyzers, thread debuggers, and static analyzers can prevent many application failures. Finally, we proposed features of execution platforms and frameworks related to process, memory, and network management that could reduce application crashes.

As future work, we aim to analyze crashes from a wider range of platforms, such as those running the iOS and Windows Mobile operating systems. Thus, we would be able to paint a broader picture around the API failures that lead mobile applications to crashes. By having data and results from the three main operating systems for mobile devices, we can validate our findings and be able to compare these systems from different angles and discuss their robustness. Additionally, by associating the stack traces from the crashes with the version of the operating system they happened, we can observe the evolution of the API design and implementation quality. Also, by analyzing metadata associated with each crash, such as the mobile operator, the network connectivity type, and the model of the mobile device where the crash was reported, we could associate crashes with particular hardware and network configurations. This will allow us to associate software failures with hardware platforms and model how particular hardware features, such as multi-core processors, affect software quality. Finally, we would like to validate our categories and recommendations by asking the mind of Android developers through questionnaires and interviews.

Acknowledgements

We would like to thank BugSense Inc., and more specifically its founders Panos Papadopoulos and John Vlachogiannis for the data and information they provided us, as well as Alexandros Kompotis, who downloaded and gave us the data set. In addition, we would like to thank Konstantinos Polychronis and Antonis Lilis for the valuable information they gave us regarding Android application development. Also, we want to thank Panos Louridas, Georgios Gousios, Marios Fragkoulis, and Vassilios Karakoidas for their suggestions and 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

Aftandilian EE, Kelley S, Gramazio C, Ricci N, Su SL, Guyer SZ (2010) Heapviz: interactive heap visualization for program understanding and debugging. In: Proceedings of the 5th international symposium on software visualization, ACM, New York, NY, USA, SOFTVIS '10, pp 53–62, doi: 10.1145/1879211.1879222

Agarwal R, Wang L, Stoller S (2006) Detecting potential deadlocks with static analysis and run-time monitoring. In: Ur S, Bin E, Wolfsthal Y (eds) Hardware and Software, Verification and Testing, Lecture Notes in Computer Science, vol 3875, Springer Berlin Heidelberg, pp 191–207, doi: 10.1007/11678779_14

Alsallakh B, Bodesinsky P, Miksch S, Nasseri D (2012) Visualizing arrays in the Eclipse Java IDE. In: Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, CSMR '12, pp 541–544, doi: 10.1109/CSMR.2012.71

Amalfitano D, Fasolino AR, Tramontana P (2011) A GUI crawling-based technique for Android mobile application testing. In: Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, IEEE Computer Society, Washington, DC, USA, ICSTW '11, pp 252–261, doi: 10.1109/ICSTW.2011.77

Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM (2012) Using GUI ripping for automated testing of Android applications. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE 2012, pp 258–261, doi: 10.1145/2351676.2351717

Artho C, Biere A (2001) Applying static analysis to large-scale, multi-threaded Java programs. In: Proceedings of the 13th Australian Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ASWEC '01, pp 68–75, doi: 10.1109/ASWEC.2001.948499

Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. Dependable and Secure Computing, IEEE Transactions on 1(1):11–33, doi: 10.1109/TDSC.2004.2

Ayewah N, Hovemeyer D, Morgenthaler J, Penix J, Pugh W (2008) Using static analysis to find bugs. Software, IEEE 25(5):22–29, doi: 10.1109/MS.2008.130

Bacchelli A (2013) Mining challenge 2013: Stack overflow. In: The 10th Working Conference on Mining Software Repositories

Bacon DF, Cheng P, Grove D (2004) Garbage collection for embedded systems. In: Proceedings of the 4th ACM international conference on Embedded software, ACM, New York, NY, USA, EMSOFT '04, pp 125–136, doi: 10.1145/1017753.1017776

Beizer B (2003) Software testing techniques. Dreamtech Press

Bloch J (2006) 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, ACM, New York, NY, USA, OOPSLA '06, pp 506–507, doi: 10.1145/1176617.1176622

Bond MD, McKinley KS (2008) Tolerating memory leaks. In: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, ACM, New York, NY, USA, OOPSLA '08, pp 109–126, doi: 10.1145/1449764.1449774

Bovet D, Cesati M (2005) Understanding The Linux Kernel. Oreilly & Associates Inc

Buse RPL, Weimer W (2012) Synthesizing API usage examples. In: Proceedings of the 2012 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE 2012, pp 782–792, doi: 10.1109/ICSE.2012.6227140

Candes E, Wakin M (2008) An introduction to compressive sampling. Signal Processing Magazine, IEEE 25(2):21–30, doi: 10.1109/MSP.2007.914731

Chang TH, Yeh T, Miller RC (2010) GUI testing using computer vision. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, New York, NY, USA, CHI '10, pp 1535–1544, doi: 10.1145/1753326.1753555

Chen G, Kandemir M, Vijaykrishnan N, Irwin MJ, Mathiske B, Wolczko M (2003) Heap compression for memory-constrained Java environments. In: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, ACM, New York, NY, USA, OOPSLA '03, pp 282–301, doi: 10.1145/949305.949330

Chen MC, Chen JL, Chang TW (2011) Android/OSGi-based vehicular network management system. Computer Communications 34(2):169–183, doi: 10.1016/j.comcom.2010.03.032

Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong MY (1992) Orthogonal defect classification—a concept for in-process measurements. Software Engineering, IEEE Transactions on 18(11):943–956

Choe H, Baek J, Jeong H, Park S (2011) MetaService: an object transfer platform between Android applications. In: Proceedings of the 2011 ACM Symposium on Research in Applied Computation, ACM, New York, NY, USA, RACS '11, pp 56–60, doi: 10.1145/2103380.2103391

Clarke S (2004) Measuring API usability. Dr Dobb's Journal 29:S6–S9, URL `http://www.drdobbs.com/windows/184405654`

Dang Y, Wu R, Zhang H, Zhang D, Nobel P (2012) ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In: Proceedings of the 2012 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE 2012, pp 1084–1093

DeMillo RA, Mathur AP (1995) A grammar based fault classification scheme and its application to the classification of the errors of TEX. Tech. rep., Citeseer

Eisenstadt M (1997) My hairiest bug war stories. Commun ACM 40(4):30–37, doi: 10.1145/248448.248456

Ellis B, Stylos J, Myers B (2007) The factory pattern in API design: a usability evaluation. In: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '07, pp 302–312

Enck W, Ongtang M, McDaniel P (2009) Understanding Android security. Security Privacy, IEEE 7(1):50–57, doi: 10.1109/MSP.2009.26

Endres A (1975) An analysis of errors and their causes in system programs. SIGPLAN Not 10(6):327–336, doi: 10.1145/390016.808455

Engler D, Ashcraft K (2003) RacerX: effective, static detection of race conditions and deadlocks. SIGOPS Oper Syst Rev 37(5):237–252, doi: 10.1145/1165389.945468

Faghri F, Bazarbayev S, Overholt M, Farivar R, Campbell RH, Sanders WH (2012) Failure scenario as a service (FSaaS) for Hadoop clusters. In: Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management, ACM, New York, NY, USA, SDMCMM '12, pp 5:1–5:6, doi: 10.1145/2405186.2405191

Farooq U, Welicki L, Zirkler D (2010) API usability peer reviews: a method for evaluating the usability of application programming interfaces. In: Proceedings of the 28th international conference on Human factors in computing systems, ACM, New York, NY, USA, CHI '10, pp 2327–2336, doi: 10.1145/1753326.1753677

Felt AP, Chin E, Hanna S, Song D, Wagner D (2011) Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security, ACM, New York, NY, USA, CCS '11, pp 627–638, doi: 10.1145/2046707.2046779

Fraboulet A, Kodary K, Mignotte A (2001) Loop fusion for memory space optimization. In: Proceedings of the 14th international symposium on Systems synthesis, ACM, New York, NY, USA, ISSS '01, pp 95–100, doi: 10.1145/500001.500025

Ganapathi A, Patterson D (2005) Crash data collection: a Windows case study. In: Proceedings of the 2005 International Conference on Dependable Systems and Networks, IEEE Computer Society, Washington, DC, USA, DSN '05, pp 280–285

Ganapathi A, Ganapathi V, Patterson DA (2006) Windows XP kernel crash analysis. In: LISA, vol 6, pp 49–159

Gavalas D, Economou D (2011) Development platforms for mobile applications: Status and trends. Software, IEEE 28(1):77–86, doi: 10.1109/MS.2010.155

Gerken J, Jetter HC, Zöllner M, Mader M, Reiterer H (2011) The concept maps method as a tool to evaluate the usability of APIs. In: Proceedings of the 2011 annual conference on Human factors in computing systems, ACM, New York, NY, USA, CHI '11, pp 3373–3382

Gray J (1986) Why do computers stop and what can be done about it? In: Symposium on reliability in distributed software and database systems, Los Angeles, CA, USA, pp 3–12

Gross KC, Bhardwaj V, Bickford R (2002) Proactive detection of software aging mechanisms in performance critical computers. In: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02), IEEE Computer Society, Washington, DC, USA, SEW '02, pp 17–23

Gross KC, Urmanov A, Votta LG, McMaster S, Porter A (2006) Towards dependability in everyday software using software telemetry. In: Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems, IEEE Computer Society, Washington, DC, USA, EASE '06, pp 9–18

Guo P, Zimmermann T, Nagappan N, Murphy B (2010) Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on, vol 1, pp 495–504, doi: 10.1145/1806799.1806871

Havelund K, Pressburger T (2000) Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer 2:366–381, doi: 10.1007/s100090050043

Havelund K, Roşu G (2004) An overview of the runtime verification tool Java PathExplorer. Form Methods Syst Des 24(2):189–215, doi: 10.1023/B:FORM.0000017721.39909.4b

Henning M (2009) API design matters. Commun ACM 52(5):46–56, doi: 10.1145/1506409.1506424

Hovemeyer D, Pugh W (2004) Finding bugs is easy. SIGPLAN Not 39(12):92–106, doi: 10.1145/1052883.1052895

Johnson PM, Kou H, Paulding M, Zhang Q, Kagawa A, Yamashita T (2005) Improving software development management through software project telemetry. IEEE Softw 22(4):76–85

Jula H, Tralamazza D, Zamfir C, Candea G (2008) Deadlock immunity: enabling systems to defend against deadlocks. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation, USENIX Association, Berkeley, CA, USA, OSDI'08, pp 295–308, URL `http://dl.acm.org/citation.cfm?id=1855741.1855762`

Jula H, Rensch T, Candea G (2011) Platform-wide deadlock immunity for mobile phones. In: Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on, pp 205 –210, doi: 10.1109/DSNW.2011.5958814

Kähkönen K, Launiainen T, Saarikivi O, Kauttio J, Heljanko K, Niemelä I (2011) LCT: an open source concolic testing tool for Java programs. In: Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'2011), Saarbrücken, Germany, pp 75–80

Kawrykow D, Robillard M (2009) Detecting inefficient API usage. In: 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009., pp 183–186, doi: 10.1109/ICSE-COMPANION.2009.5070977

Kim D, Wang X, Kim S, Zeller A, Cheung S, Park S (2011a) Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. Software Engineering, IEEE Transactions on 37(3):430–447, doi: 10.1109/TSE.2011.20

Kim H, Lee M, Han W, Lee K, Shin I (2011b) Aciom: application characteristics-aware disk and network I/O management on Android platform. In: Proceedings of the ninth ACM international conference on Embedded software, ACM, New York, NY, USA, EMSOFT '11, pp 49–58, doi: 10.1145/2038642.2038652

Kim H, Agrawal N, Ungureanu C (2012) Revisiting storage for smartphones. ACM Trans on Storage 8(4):14:1–14:25, doi: 10.1145/2385603.2385607

Kim K (2006) A non-blocking buffer mechanism for real-time event message communication. Real-Time Systems 32:197–211, doi: 10.1007/s11241-005-4680-7

Kim S, Zimmermann T, Nagappan N (2011c) Crash graphs: an aggregated view of multiple crashes to improve crash triage. In: Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks, IEEE Computer Society, Washington, DC, USA, DSN '11, pp 486–493

Knuth DE (1989) The errors of tex. Software: Practice and Experience 19(7):607–685, doi: 10.1002/spe.4380190702

Lee I, Iyer R (1995) Software dependability in the tandem guardian system. Software Engineering, IEEE Transactions on 21(5):455–467, doi: 10.1109/32.387474

Li Z, Tan L, Wang X, Lu S, Zhou Y, Zhai C (2006) Have things changed now?: An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st workshop on Architectural and system support for improving software dependability, ACM, New York, NY, USA, ASID '06, pp 25–33

Liblit B, Aiken A (2002) Building a better backtrace: techniques for postmortem program analysis. Tech. rep., Berkeley, Berkeley, CA, USA

Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Di Penta M, Oliveto R, Poshyvanyk D (2013) API change and fault proneness: A threat to the success of Android apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2013, pp 477–487, doi: 10.1145/2491411.2491428

Long B, Hoffman D, Strooper P (2003) Tool support for testing concurrent Java components. IEEE Trans Softw Eng 29(6):555–566, doi: 10.1109/TSE.2003.1205182

Maalej W, Robillard MP (2013) Patterns of knowledge in API reference documentation. IEEE Transactions on Software Engineering 99(PrePrints):1, doi: 10.1109/TSE.2013.12

Maia C, Nogueira LM, Pinho LM (2010) Evaluating Android OS for embedded real-time systems. In: Petters SM, Zijlstra P (eds) 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010), Politécnico do Porto, pp 63–70

Maji AK, Hao K, Sultana S, Bagchi S (2010) Characterizing failures in mobile OSes: a case study with Android and Symbian. In: Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering, IEEE Computer Society, Washington, DC, USA, ISSRE '10, pp 249–258

Mandelin D, Xu L, Bodík R, Kimelman D (2005) Jungloid mining: helping to navigate the API jungle. SIGPLAN Not 40(6):48–61, doi: 10.1145/1064978.1065018

McKinley KS, Carr S, Tseng CW (1996) Improving data locality with loop transformations. ACM Trans Program Lang Syst 18(4):424–453, doi: 10.1145/233561.233564

Meijer E, Beckman B, Bierman G (2006) LINQ: reconciling object, relations and XML in the .NET framework. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM, New York, NY, USA, SIGMOD '06, pp 706–706, doi: 10.1145/1142473.1142552

Mernik M, Heering J, Sloane AM (2005) When and how to develop domain-specific languages. ACM Comput Surv 37(4):316–344, doi: 10.1145/1118890.1118892

van der Merwe H, van der Merwe B, Visse W (2012) Verifying Android applications using Java PathFinder. SIGSOFT Softw Eng Notes 37(6):1–5, doi: 10.1145/2382756.2382797

Michael MM, Scott ML (1996) Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, ACM, New York, NY, USA, PODC '96, pp 267–275, doi: 10.1145/248052.248106

Ongtang M, Butler K, McDaniel P (2010) Porscha: policy oriented secure content handling in Android. In: Proceedings of the 26th Annual Computer Security Applications Conference, ACM, New York, NY, USA, ACSAC '10, pp 221–230, doi: 10.1145/1920261.1920295

Panda PR, Catthoor F, Dutt ND, Danckaert K, Brockmeyer E, Kulkarni C, Vandercappelle A, Kjeldsberg PG (2001) Data and memory optimization techniques for embedded systems. ACM Trans Des Autom Electron Syst 6(2):149–206, doi: 10.1145/375977.375978

Payet T, Spoto F (2012) Static analysis of Android programs. Inf Softw Technol 54(11):1192–1201, doi: 10.1016/j.infsof.2012.05.003

Pheatt C (2008) Intel threading building blocks. J Comput Sci Coll 23(4):298–298, URL `http://dl.acm.org/citation.cfm?id=1352079.1352134`

Ploski J, Rohr M, Schwenkenberg P, Hasselbring W (2007) Research issues in software fault categorization. ACM SIGSOFT Softw Eng Notes 32(6), doi: 10.1145/1317471.1317478

Podgurski A, Leon D, Francis P, Masri W, Minch M, Sun J, Wang B (2003) Automated support for classifying software failure reports. In: 25th International Conference on Software Engineering, 2003. Proceedings., IEEE Computer Society, Washington, DC, USA, pp 465–475, doi: 10.1109/ICSE.2003.1201224

Robillard M, DeLine R (2011) A field study of API learning obstacles. Empirical Software Engineering 16(6):703–732, doi: 10.1007/s10664-010-9150-8

Robillard M, Bodden E, Kawrykow D, Mezini M, Ratchford T (2013) Automated API property inference techniques. Software Engineering, IEEE Transactions on 39(5):613–637, doi: 10.1109/TSE.2012.63

Robillard MP (2009) What makes APIs hard to learn? Answers from developers. IEEE Softw 26(6):27–34, doi: 10.1109/MS.2009.193

Schoeberl M (2004) Restrictions of Java for embedded real-time systems. In: Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2004. Proceedings., pp 93 –100, doi: 10.1109/ISORC.2004.1300334

Sen K, Agha G (2006) CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: In CAV, Springer, pp 419–423

Seo J, Choi B, Yang S (2011) A profiling method by PCB hooking and its application for memory fault detection in embedded system operational test. Information and Software Technology 53(1):106 – 119, doi: 10.1016/j.infsof.2010.09.003

Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S, Glezer C (2010) Google Android: a comprehensive security assessment. Security Privacy, IEEE 8(2):35–44, doi: 10.1109/MSP.2010.2

Shabtai A, Kanonov U, Elovici Y, Glezer C, Weiss Y (2012) "Andromaly": a behavioral malware detection framework for Android devices. J Intell Inf Syst 38(1):161–190, doi: 10.1007/s10844-010-0148-x

Shelton C, Koopman P, Devale K (2000) Robustness testing of the Microsoft Win32 API. In: Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on, pp 261–270, doi: 10.1109/ICDSN.2000.857548

Shi L, Zhong H, Xie T, Li M (2011) An empirical study on evolution of API documentation. In: Giannakopoulou D, Orejas F (eds) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol 6603, Springer Berlin Heidelberg, pp 416–431, doi: 10.1007/978-3-642-19811-3_29

Shihab E, Kamei Y, Bhattacharya P (2012) Mining challenge 2012: The android platform. In: The 9th Working Conference on Mining Software Repositories

de Souza C, Bentolila D (2009) Automatic evaluation of API usability using complexity metrics and visualizations. In: 31st International Conference on Software Engineering - Companion

Volume, 2009. ICSE-Companion 2009., pp 299–302, doi: 10.1109/ICSE-COMPANION.2009.5071006

Sproull R, Waldo J (2014) The api performance contract. Queue 12(1):10:10–10:20, doi: 10.1145/2576966.2576968

Stylos J (2009) Making APIs More Usable with Improved API Designs, Documentation and Tools. Carnegie Mellon University, URL `http://books.google.co.uk/books?id=MQYoWvOnsy8C`

Stylos J, Clarke S (2007) Usability implications of requiring parameters in objects' constructors. In: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '07, pp 529–539, doi: 10.1109/ICSE.2007.92

Stylos J, Myers BA (2008) The implications of method placement on API learnability. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, ACM, New York, NY, USA, SIGSOFT '08/FSE-16, pp 105–112, doi: 10.1145/1453101.1453117

Sullivan M, Chillarege R (1991) Software defects and their impact on system availability—a study of field failures in operating systems. In: Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium, pp 2–9, doi: 10.1109/FTCS.1991.146625

Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2013) Bug characteristics in open source software. Empirical Software Engineering pp 1–41, doi: 10.1007/s10664-013-9258-8

Tulach J (2012) Practical API Design: Confessions of a Java Framework Architect. Apressus Series, Apress, URL `http://books.google.co.uk/books?id=5DmYpwAACAAJ`

Vallina-Rodriguez N, Crowcroft J (2013) Energy management techniques in modern mobile handsets. Communications Surveys Tutorials, IEEE 15(1):179–198, doi: 10.1109/SURV.2012.021312.00045

Valois JD (1995) Lock-free linked lists using compare-and-swap. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, ACM, New York, NY, USA, PODC '95, pp 214–222, doi: 10.1145/224964.224988

Vaughan-Nichols SJ (2001) Technology news. Computer 34(12):22–24, doi: 10.1109/2.970549

Xie T, Pei J (2006) MAPO: mining API usages from open source repositories. In: Proceedings of the 2006 international workshop on Mining software repositories, ACM, New York, NY, USA, MSR '06, pp 54–57

Yang L, Dick RP, Lekatsas H, Chakradhar S (2010) Online memory compression for embedded systems. ACM Trans Embed Comput Syst 9(3):27:1–27:30, doi: 10.1145/1698772.1698785

Yang T, Hertz M, Berger ED, Kaplan SF, Moss JEB (2004) Automatic heap sizing: taking real memory into account. In: Proceedings of the 4th international symposium on Memory management, ACM, New York, NY, USA, ISMM '04, pp 61–72, doi: 10.1145/1029873.1029881

Appendix

Here we provide examples of cleaned stack traces that cover cases from the *formulae* in Section 3.3.

Table 13: Example of Case 4.

F: Framework	dalvik.system.NativeStart.main com.android.internal.os.ZygoteInit.main com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run java.lang.reflect.Method.invoke java.lang.reflect.Method.invokeNative android.app.ActivityThread.main android.os.Looper.loop android.os.Handler.dispatchMessage android.os.Handler.handleCallback android.view.View$PerformClick.run android.view.View.performClick
A: Application	com.example.TabsFragmentActivity$2.onClick
I: API	android.widget.CompoundButton.setChecked android.widget.RadioGroup$CheckedStateTracker.onCheckedChanged android.widget.RadioGroup.access$600 android.widget.RadioGroup.setCheckedId
A: Application	com.example.TabsFragmentActivity$4.onCheckedChanged
I: API	android.widget.TabHost.setCurrentTab android.widget.TabHost.invokeOnTabChangeListener
A: Application	com.example.TabManager.onTabChanged
I: API	android.support.v4.app.FragmentManagerImpl.executePendingTransactions android.support.v4.app.FragmentManagerImpl.execPendingActions android.support.v4.app.BackStackRecord.run android.support.v4.app.FragmentManagerImpl.attachFragment android.support.v4.app.FragmentManagerImpl.moveToState
A: Application	com.example.DiscussionListFragment.onViewCreated
I: API	android.widget.ListView.setAdapter
A: Application	com.example.DiscussionAdapter.getCount
E: Exception	!java.lang.NullPointerException

Table 14: Example of Case 5.

F: Framework	dalvik.system.NativeStart.main
	com.android.internal.os.ZygoteInit.main
	com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run
	java.lang.reflect.Method.invoke
	java.lang.reflect.Method.invokeNative
	android.app.ActivityThread.main
	android.os.Looper.loop
	android.os.Handler.dispatchMessage
	android.view.ViewRoot.handleMessage
	com.android.internal.policy.impl.PhoneWindow$-DecorView.dispatchTouchEvent
	android.app.Activity.dispatchTouchEvent
	com.android.internal.policy.impl.PhoneWindow.superDispatchTouchEvent
	com.android.internal.policy.impl.PhoneWindow$-DecorView.superDispatchTouchEvent
	android.view.ViewGroup.dispatchTouchEvent
	android.view.ViewGroup.dispatchTouchEvent
	android.view.ViewGroup.dispatchTouchEvent
	android.view.ViewGroup.dispatchTouchEvent
	android.view.ViewGroup.dispatchTouchEvent
	android.view.View.dispatchTouchEvent
A: Application	com.example.RepeatingImageButton.onTouchEvent
I: API	android.view.View.onTouchEvent
	android.view.View.performClick
A: Application	com.example.ControlBar$4.onClick
	com.example.MediaService$ServiceStub.play
	com.example.MediaService.play
	com.example.MediaUtils.getGoodArtwork
	com.example.MediaUtils.getArtwork
	com.example.MediaUtils.getDefaultArtwork
I: API	android.graphics.BitmapFactory.decodeStream
	android.graphics.BitmapFactory.nativeDecodeAsset
E: Exception	!java.lang.OutOfMemoryError

In the following graphs, we present the diagrams for the distribution of the signatures (total and unique frequencies) among the crash cause categories for each of the examined APIs.

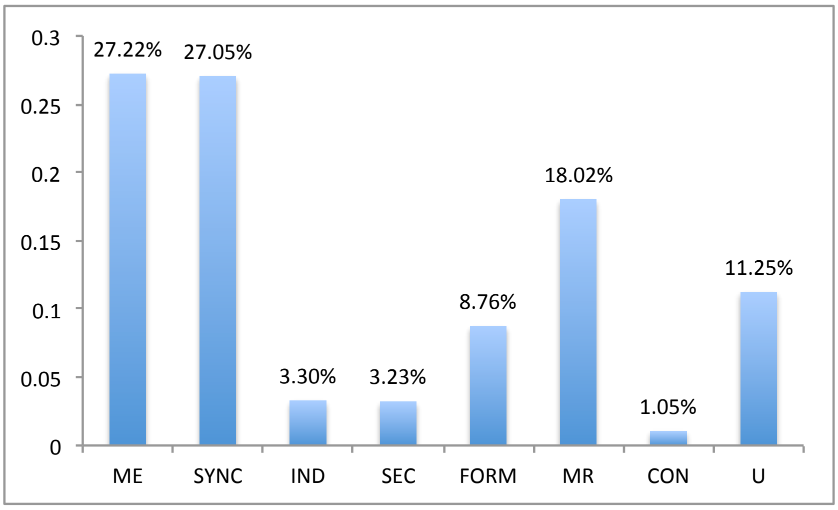

Fig. 6: Total signatures for Android APIs.

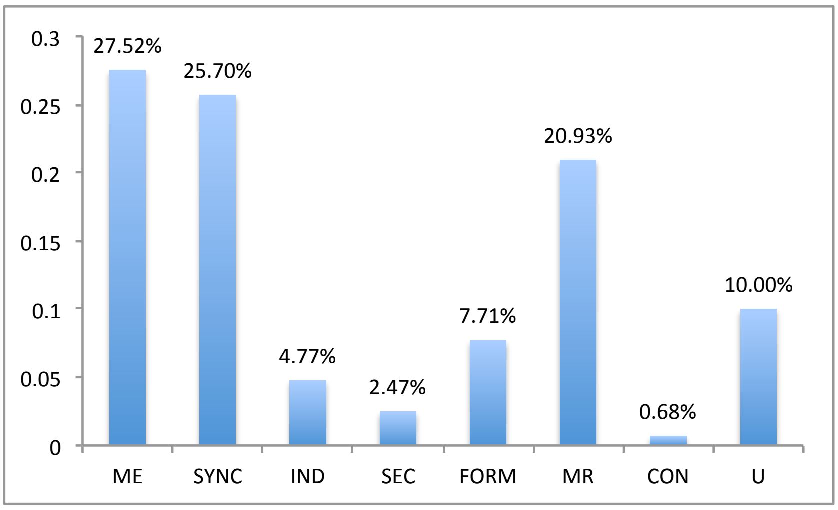

Fig. 7: Unique signatures for Android APIs.

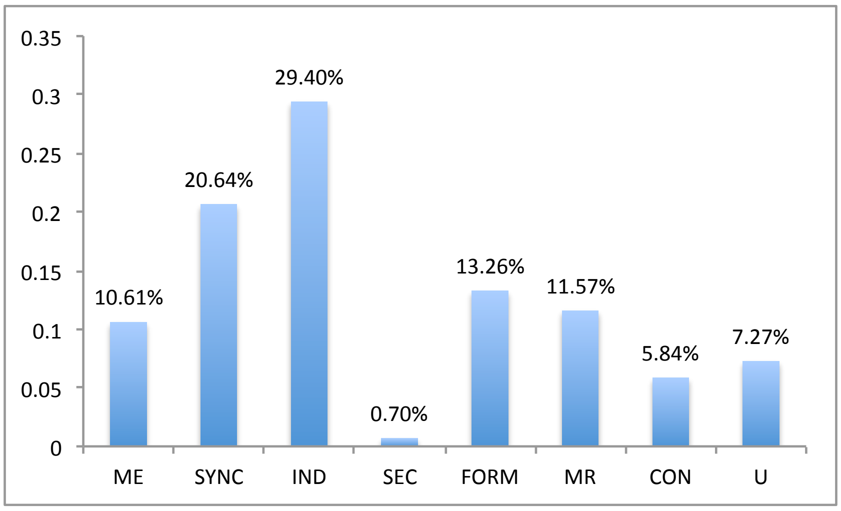

Fig. 8: Total signatures for Java APIs.

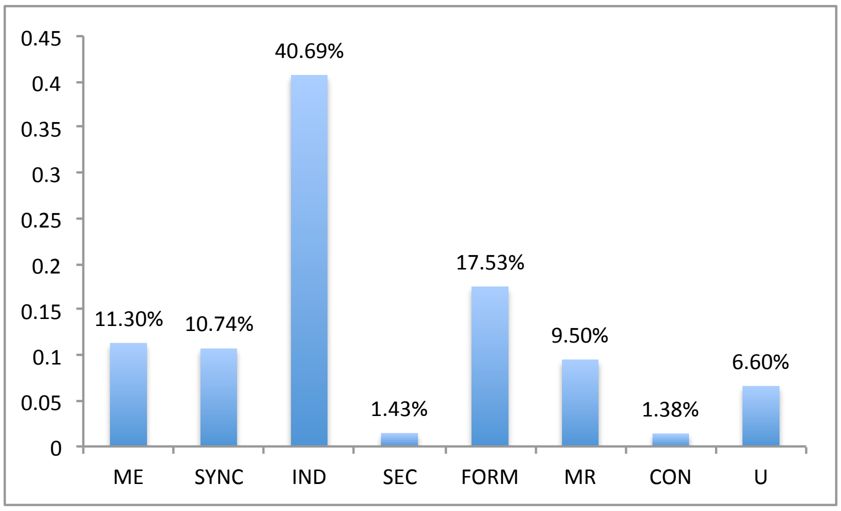

Fig. 9: Unique signatures for Java APIs.

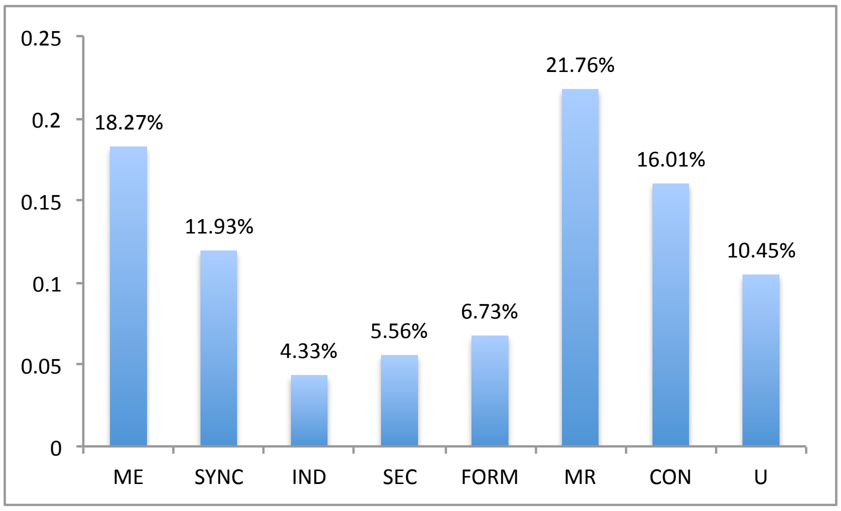

Fig. 10: Total signatures for third-party library APIs.

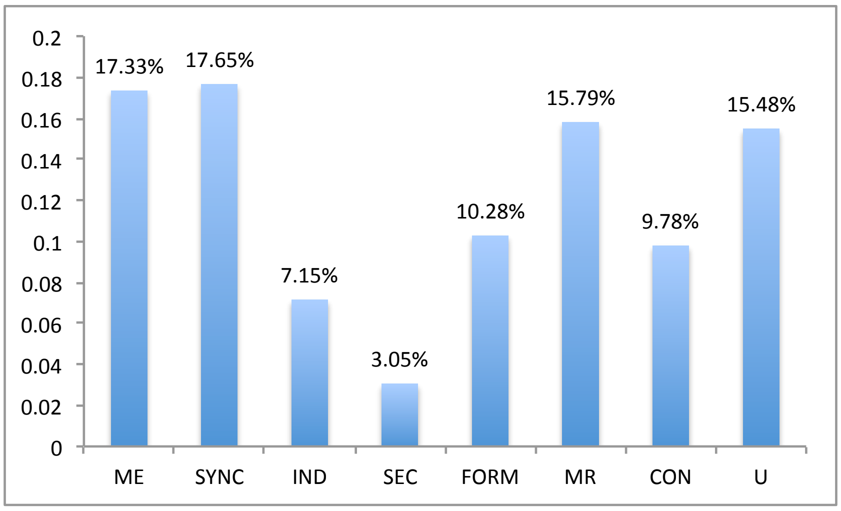

Fig. 11: Unique signatures for third-party library APIs.